60482700

CONTROL DATA CORPORATION

CYBER INTERACTIVE DEBUG VERSION 1 GUIDE FOR USERS OF FORTRAN EXTENDED VERSION 4

CDC® OPERATING SYSTEMS:
NOS 1
NOS/BE 1

REVISION RECORD	
REVISION	**DESCRIPTION**
A	Original release at PSR level 472.
(2-1-79)	
Publication No. 60482700	

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
P. O. BOX 3492
SUNNYVALE, CALIFORNIA 94088-3492

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Cover	–
Title Page	–
ii	A
iii/iv	A
v/vi	A
vii thru ix	A
1-1 thru 1-4	A
2-1 thru 2-11	A
3-1 thru 3-31	A
4-1 thru 4-10	A
5-1 thru 5-23	A
6-1 thru 6-5	A
A-1 thru A-3	A
B-1	A
B-2	A
C-1 thru C-3	A
D-1 thru D-3	A
E-1	A
E-2	A
Index-1	A
Index-2	A
Comment Sheet	A
Mailer	A
Back Cover	–

Page	Revision

Page	Revision

PREFACE

This manual is intended to provide the FORTRAN programmer with assistance in the debugging of FORTRAN Extended programs under the control of the CDC® CYBER Interactive Debug Facility.

CYBER Interactive Debug (CID) operates under the following operating systems:

NOS/BE 1 for the CONTROL DATA® CYBER 170 Series; CYBER 70 Models 71, 72, 73, 74; and 6000 Series Computer Systems.

NOS 1 for the CDC® CYBER 170 Models 171, 172, 173, 174, 175; CYBER 70 Models 71, 72, 73, 74; and 6000 Series Computer Systems.

You should have a copy of the CYBER Interactive Debug reference manual available for reference, but you need not be familiar with the manual. In addition, you should be familiar with FORTRAN Extended and should be able to run jobs interactively under either NOS/BE INTERCOM or the NOS Time-Sharing System.

This manual provides a tutorial approach to CID beginning with basic features and proceeding through more advanced features. It is not comprehensive in its approach to CID; only those features considered useful to FORTRAN programmers are discussed. Most of the features discussed in this manual are illustrated by actual examples of debug sessions. This is intended to help you become familiar with CID notational conventions and with information produced by CID.

Additional information can be found in the publications listed below.

Publication	Publication Number
CYBER Interactive Debug Version 1 Reference Manual	60481400
FORTRAN Extended Version 4 Reference Manual	60497800
INTERCOM Guide for Users of FORTRAN Extended 4	60495000
NOS Time-Sharing User's Guide	60436400

CDC manuals can be ordered from Control Data Corporation Literature and Distribution Services, 8001 East Bloomington Freeway, Minneapolis, MN 55420.

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

CONTENTS

APPENDIXES

FIGURES

TABLES

INTRODUCTION 1

The CYBER Interactive CID Facility (CID) provides you with the capability of interactively debugging an executing object program. CID can be used with FORTRAN Extended programs compiled under the NOS or NOS/BE operating systems.

Debug mode is established by means of a control statement. As long as debug mode is in effect, execution of all user programs takes place under control of CID. CID, in turn, allows you to enter commands that perform the following operations:

- Suspend program execution at specified locations.
- Suspend program execution on the occurrence of selected conditions, such as modification of a variable.
- Display the contents of variables, arrays, and common blocks while execution is suspended.
- Change the contents of variables, arrays, or common blocks within the program while execution is suspended.
- Resume program execution at the location where it was suspended or at another location.

WHAT IS INTERACTIVE DEBUGGING?

Interactive debugging means that you debug your program while it is executing. In interactive mode, CID allows you to suspend execution of your program and enter commands directly from a terminal while execution is suspended. CID executes each command immediately after it is entered. Program execution remains suspended until resumed by the appropriate command. In this manner, you can control and monitor the execution of your program, stopping at desired points to examine and modify the values of program variables.

DESCRIPTION OF A DEBUG SESSION

A significant characteristic of CID is that much of its power exists in a few commands. It is not necessary to have a complete knowledge of all the CID commands to take advantage of the most powerful features of CID.

Following is a step-by-step summary of a basic debug session that should provide a useful debugging tool. The commands and terms used in this summary are discussed in greater detail in sections 2 through 6.

To use CID:

1. Type DEBUG to turn on debug mode.
2. Compile and load your program normally. Control transfers to CID when execution begins. CID displays a message at the terminal and waits for your input.
3. Set traps and breakpoints as desired.

   To set a breakpoint at a line number or statement label enter:

   ```
   SET,BREAKPOINT,P.name_L.n
   SET,BREAKPOINT,P.name_S.n
   ```

   where name is a main program, subroutine, or function name and n is a line number or statement label.

   To set a STORE trap enter:

   ```
   SET,TRAP,STORE,P.name_variable
   ```

   where name is the name of the program unit containing the specified variable. This trap suspends program execution whenever the specified variable is modified.
4. To begin execution of your program enter GO.
5. CID executes your program normally, but returns control to you when a trap or breakpoint occurs.

   At this point you can display the values of program variables with the statements:

   ```
   PRINT*,variable list
   DISPLAY,P.name_variable
   ```

   You can remove existing traps or breakpoints with the commands:

   ```
   CLEAR,BREAKPOINT,P.name_L.n
   CLEAR,BREAKPOINT,P.name_S.n
   CLEAR,TRAP,STORE
   ```

   To resume execution enter GO.
6. To terminate the session enter QUIT. To turn off debug mode enter DEBUG(OFF).

SPECIAL FORTRAN EXTENDED FEATURES

CID provides certain features currently available only to FORTRAN Extended programs compiled in debug mode. These features include commands with a FORTRAN-like syntax and the capability of referencing locations within an object program by statement label, line number, or variable name. The commands available only in debug mode are indicated in appendix E.

For purposes of this user's guide, it is assumed that a FORTRAN program to be executed under CID control will be compiled in debug mode; therefore, no distinction is made between standard features and the special FORTRAN features in the discussions of the CID capabilities.

WHY USE CID

Conventional debugging techniques often require the use of load maps, object listings, and octal dumps. In addition, it is often necessary to recompile a FORTRAN program several times to make corrections or to add statements that print intermediate values of program variables. These debugging techniques can be expensive in terms of both machine time and programmer time.

CID, however, requires only a source listing. CID commands allow you to debug your program directly from the source listing, referencing variables and line numbers symbolically. In many cases a FORTRAN program need be compiled only once; the resulting object program can be executed repeatedly with different CID commands specified for each run. Since CID allows you to make changes to your program's data and control flow as execution proceeds, you can often accomplish in a single session debugging that would normally require several compilations. Thus, considerable time savings can be realized, especially when debugging programs that are time-consuming to compile or execute.

A disadvantage of CID is that TS compilation mode is required if the special FORTRAN commands and symbolic capabilities are to be used. Since a program that executes correctly when compiled in TS mode might not do so when compiled in a higher mode of optimization, a program tested in TS mode should also be tested with OPT=1 or OPT=2. If the program does not execute correctly in optimizing mode, then you must either use conventional debugging techniques or use CID without the FORTRAN capabilities.

COMMON PROGRAMMING ERRORS

The following paragraphs describe some of the common mistakes committed by FORTRAN programmers. These errors often lead to execution time errors called mode errors that result in abnormal termination of execution. CID has a feature that transfers control to CID when abnormal termination occurs, allowing you to enter commands to determine the cause of the error. Mode errors are discussed in appendix C.

INDEXING

A frequent source of execution time errors is faulty indexing, especially when the indexing occurs within a DO loop. This usually involves a subscript that exceeds the dimensioned boundary of an array. Incorrect indexing can cause the program to reference inaccessible locations, which can have unpredictable effects on program execution. If an illegal index results in a reference to a location outside the program's field length, a mode 1 (address out of range) error occurs. If the reference is within the program's field length it can result in the overstoring of data or program instructions, or it can result in the use of invalid data in subsequent computations.

CID allows you to display the contents of program variables and to observe interactively the behavior of array subscripts to check for array boundary errors. A special form of the PRINT command displays a warning message when an index exceeds an array boundary. A familiarity with machine representations of numbers can help you recognize invalid data. Machine representations are described in appendix C.

SUBPROGRAM CALLS

Calls to nonexistent subprograms or calls with an incorrect number of arguments are common errors. These errors can cause the user program to reference locations outside the allowed field length or to unintentionally overwrite areas within the field length.

You can debug programs containing subprogram calls by using a CID feature called an RJ trap which suspends program execution immediately prior to executing a function or subroutine call and immediately prior to executing a RETURN. CID displays the program unit name and line number where the CALL or RETURN occurs. You can then specify commands to display the status of the program as it exists at that precise moment.

INITIALIZATION

Failing to set a variable to its proper value before use is a common source of error. An undefined variable can produce unexpected results when used in subsequent computations. (The value assigned to uninitialized sections of core is an installation parameter. In some cases, a special indefinite value is used which causes abnormal termination when used as an operand in a subsequent computation.)

You can determine if a variable has been properly initialized by issuing CID commands to display its value. Machine representations of operands are shown in appendix C.

ARITHMETIC ERRORS

Arithmetic errors occur when an attempt is made to perform an illegal arithmetic operation or when a result is generated which exceeds the capacity of the central processor. Such errors can be caused by dividing a number by zero or by performing arithmetic operations on very large or very small numbers. When these conditions occur, an infinite operand is usually generated which causes abnormal termination of execution when used in a subsequent computation. Arithmetic errors can also produce indefinite operands (appendix C).

You can use CID to display the contents of program variables to determine if they contain numbers that might lead to arithmetic errors. An infinite value is displayed as the character R (for out-of-range). An indefinite is displayed as the character I. A knowledge of machine capacities can be helpful. The maximum and minimum allowable numbers are presented in appendix C.

FLOATING POINT ERRORS

Errors in floating point computations often occur because certain floating point numbers cannot be exactly represented in a 60-bit word. Such numbers must be represented by an approximation. For example, the decimal fraction .1 does not have an exact binary representation. Thus, in the program segment:

```
      A=0.0
      DO 10 1=1, 10
10    A=A+.1
```

the final value of A is not 1.0, as you might expect, but .9999....

As another example, the computation:

```
(1./3.)*3.
```

clearly has a true value of 1.0. When performed on a binary processor, however, the result is .999..., because 1./3. is an infinitely repeating fraction and cannot be precisely represented.

Such errors might seem trivial, but when involved in successive computations, the cumulative effect can become significant.

Because of the computer's inability to exactly represent certain numbers, you can get into trouble when testing floating point values for equality, as in the statement:

```
IF (X.EQ.1.0) GO TO 2
```

If the value stored in X is a calculated value, then the preceding condition might never be satisfied. This statement should be replaced by:

```
IF (X-1.0.LT.D) GO TO 20
```

where D is a value indicating the desired degree of accuracy. CID helps to reveal this type of error by allowing you to display variable values during program execution.

INPUT CHECKING

An important aspect of debugging is the checking of input data. Many errors occur because a program does not handle all possiblities for input data, such as end points, extreme values, or the case where no data, or less than the expected amount, is supplied. If certain input values are invalid, the program should check for those values. Test cases should be designed to include all possibilities for input data.

CID provides a feature (described in section 5) that allows you to simulate the reading of input data by specifying a sequence of commands to be executed in place of the READ statement. With this feature, you can design and run test cases without the necessity of creating separate data files.

PROGRAMMING FOR EASE OF DEBUGGING

When coding a FORTRAN program, there are certain guidelines you can follow to make debugging easier. Probably the most important of these is program modularity. Simply stated, program modularity means limiting the size of program units and dividing programs into subprograms that perform a logical function. A modular program is easier to understand, easier to modify, and easier to debug.

CID lends itself to use with a modular program. Through CID, you can gain control on entry into and exit from a subprogram. You can use CID to display values input to a subprogram, intermediate values used in computations within the subprogram, and values output from the subprogram. By specifying special parameters on CID commands, you can restrict the scope of the commands to particular program units.

Using a style of coding that avoids GO TO's and minimizes branches can be an aid in the debugging process. A program that contains a minimum of branches and flows logically from top to bottom is much easier to understand than one that contains many needless branches. CID provides features that allow you to trace the flow of control of your executing program; this process is much easier if the program avoids needlessly complex logic. (Refer to the FORTRAN Extended user's guide for a discussion of top down coding.)

Finally, you should avoid programming tricks and shortcuts, particularly if they depend on system idiosyncracies. For example, although some systems initialize memory to zero, it is best to include code in your program which performs all appropriate initialization.

CID should not be considered a substitute for proper programming practices. Even though CID offers many powerful features, a well-written program is much easier to debug.

Program carefully and try to minimize the number of errors. Performing a careful visual scan of the program before execution can reveal many of the more obvious errors. It is better to have correct code to begin with than to spend time debugging.

WHAT EFFECT DOES CID HAVE ON PROGRAM SIZE AND EXECUTION TIME?

If the special FORTRAN features are to be used in a debug session, the program must be compiled in debug mode. This requires TS compilation mode. TS compilation generates unoptimized object code, generally resulting in faster compilation but slower execution. The minimum field length requirement for a program compiled in TS mode is 40000_8 words. In addition, compiling in debug mode generates additional code for use by CID.

The CID module, which is loaded into the user's field length, increases the memory requirement by approximately 4000 words. Programs that become excessively large should be modularized, and the modules debugged separately.

Certain CID features require a mode of execution called interpret mode (described in section 3). Execution in interpret mode can require up to 50 times more computer time than normal execution. To reduce execution time, CID provides a command that can be used to turn interpret mode off while executing portions of a program already debugged. If necessary, your execution time limit can be increased by the ETL control statement (NOS/BE) or the SETTL control statement (NOS).

If a debug session exceeds its allotted execution time, a time limit interrupt occurs and CID issues the message:

```
*TIME LIMIT
```

This frequently occurs while executing in interpret mode. When a time limit interrupt occurs under NOS, you have the option to increase the time limit and resume the session at the point of the interrupt. This is done by entering T,n where n is an octal number of CPU seconds. The best course of action to avoid a time limit interrupt is to examine the debug session to determine ways to speed it up. Usually, there are several ways of using CID to accomplish a particular task. Following are some suggestions for streamlining debug sessions:

- Use the CLEAR,INTERPRET command when in interpret mode and executing portions of a program not requiring interpret mode.

- Substitute breakpoints for traps that require interpret mode. You should not be discouraged from using interpret mode, but you should be aware of its time requirements.

- Use the frequency parameters (discussed in section 3) when setting breakpoints within DO loops to cut down on unnecessary suspensions of execution.

- Avoid using command files if time is critical. Use of these is time-consuming since it increases the number of reads. You should not be discouraged from using command files (discussed in section 5), but you should be aware of their time requirements.

- Always perform a careful desk-check of your program before initiating a debug session; use other available debugging aids, such as the FORTRAN XREF. Do not rely on CID as your sole debugging tool.

- Execution of command sequences can be time-consuming; try to keep them short and simple.

- Don't try to perform too many operations in a single debug session. If necessary, recompile your program, correcting known bugs, and conduct additional sessions.

OVERLAY DEBUGGING

CID can be used with programs containing overlays. CID provides features intended specifically for the debugging of programs with overlays, including a special trap that allows you to suspend execution of an object program when an overlay is loaded. Overlay debugging is discussed in section 6.

CID cannot be used with programs loaded by either SEGLOAD or the user-call loader.

BATCH MODE DEBUGGING

Although CID is mainly intended to be used interactively, it can be used in batch mode. Batch mode debugging is discussed in appendix D.

THE FORTRAN EXTENDED DEBUGGING FACILITY

The FORTRAN Extended Debugging Facility (described in the FORTRAN Extended reference manual) can be used to debug programs executed in batch mode. The debugging facility consists of special statements that are inserted into a source program and processed at compile time. When the program is executed the debugging statements produce output that can help the user find errors in his program. Some of the functions of the debugging facility are similar to those of CID. These include the capability of checking subroutine calls and returns, values stored into variables, and the flow of execution. In addition, the debugging facility has capabilities that are not provided by CID, such as the ability to check array bounds.

The debugging facility does have some disadvantages, however. Since it can be used only in batch mode, a debugging session must be completely planned in advance of program execution. To make changes to debugging statements or to the program itself it is necessary to recompile the program. CID, on the other hand, allows you to enter debugging statements interactively, and multiple debug sessions can be conducted without recompiling. As this manual should help to make clear, CID has many additional capabilities that the debugging facility does not have.

Execution time is comparable using either CID or the debugging facility unless CID is executed in interpret mode, which can greatly increase execution time.

This section summarizes the operations necessary for conducting a debug session and introduces some CID notation conventions. At the end of the section some basic commands are presented and used in a sample session. These commands will enable you to conduct a productive debug session.

ENTERING THE DEBUG ENVIRONMENT

To execute a program under CID control (and to make use of the FORTRAN capabilities) you must compile and execute the program in debug mode.

There are two ways to compile a program in debug mode:

- Initiate debug mode prior to compilation with the DEBUG control statement.
- Specify the DB parameter on the FTN control statement.

To execute the program under CID control you must initiate debug mode prior to the program load.

DEBUG CONTROL STATEMENT

The DEBUG control statement activates debug mode. The format of this statement is:

```
DEBUG
```

or

```
DEBUG(ON)
```

When debug mode is on, you can enter system control statements in a normal manner. However, when a FORTRAN program is compiled in debug mode a symbol table and a line number table are generated as part of the object code. CID uses these tables while processing the object program to determine variable locations, source line locations, and statement locations.

When a program that has been compiled in debug mode is subsequently executed in debug mode, all the CID features can be used. Note that a program that has not been compiled in debug mode can still be executed in debug mode but the special FORTRAN commands cannot be used and program locations cannot be referenced symbolically.

If you are using the FTNTS subsystem (NOS) or the INTERCOM EDITOR (NOS/BE), you can compile and execute in debug mode by specifying the CID control statement prior to the first RUN command. Note that under NOS, the DEBUG statement, as with all system control statements, must be entered while in the batch subsystem.

The statement to deactivate debug mode is:

```
DEBUG(OFF)
```

When debug mode is turned off, programs compiled in debug mode execute normally.

DB PARAMETER

Including the DB parameter on the FTN control statement has the same result as compiling in debug mode. In most cases, it will not be necessary to use this parameter. It is generally used when compiling in batch mode. The DB parameter automatically activates TS mode. For example:

```
FTN,I=PROGA,DB,L=LIST
```

compiles the source program in file PROGA, generates CID tables, and writes the output listing to file LIST.

EXECUTING UNDER CID CONTROL

A debug session consists of the sequence of interactions between you and CID which takes place while your object program is executing in debug mode. The session begins when you initiate execution of your object program and ends when you enter the QUIT command.

If you are executing under the NOS/BE EDITOR or the NOS FTNTS subsystem, you can begin the session by issuing the appropriate RUN command, since this command automatically initiates program execution after compilation is complete. If your program was compiled with an FTN control statement the session is initiated by entering the name of the binary object file (default name is LGO). The system loads the CID program module along with your binary program and system and library modules. Control then transfers to an entry point in CID. CID then issues the message:

```
CYBER INTERACTIVE DEBUG
?
```

The ? character is a prompt signifying that CID is waiting for user input. At this point you can enter CID commands.

The examples in figure 2-1 show the statements necessary for compiling a program and initiating a debug session under the NOS and NOS/BE operating systems.

Debugging a program might require more than one debug session. If this is the case, you can terminate the current session, rewind the binary file, and initiate a new session, as in the following NOS/BE example:

```
?quit
DEBUG TERMINATED
..rewind,lgo
..lgo
CYBER INTERACTIVE DEBUG
?
```

Example 1: Compilation Under NOS/BE INTERCOM Editor.

```
..edit,proga,seq

..debug

..run,ftn

   41000B   CM STORAGE USED
       .041 CP SECONDS COMPILATION TIME

CYBER INTERACTIVE DEBUG
?
```

Example 2: Compilation Under NOS/BE INTERCOM.

```
COMMAND- debug

COMMAND- ftn,i=proga,l=list

   41000B   CM STORAGE USED
       .044 CP SECONDS COMPILATION TIME
COMMAND- lgo

CYBER INTERACTIVE DEBUG
?
```

Example 3: Compilation Under NOS Time-Sharing System.

```
/debug
 $DEBUG.
/ftnts
 OLD, NEW, OR LIB FILE: old,proga


  READY.
 run

  78/12/12. 08.07.41.
 PROGRAM   PROGA

  CYBER INTERACTIVE DEBUG
 ?
```

Figure 2-1. Initiating a Debug Session

Under NOS:

```
?quit
SRU        3.266 UNTS.
RUN COMPLETE.
batch
$RFL,0.
/rewind,lgo
$REWIND,LGO
/lgo
CYBER INTERACTIVE DEBUG
?
```

Any CID commands issued during a session apply only to that session; subsequent sessions revert to the original compiled version of the program. Note that even though a debug session has been terminated, debug mode remains on until you turn it off with DEBUG(OFF). When CID is on, all executions of user programs occur under CID control.

ENTERING CID COMMANDS

The CID prompt for user response is a ? character. You enter a CID command on the same line and press RETURN. CID then processes the command, issues an informative message indicating the disposition of the command or displays any output that the command calls for, and issues another ? prompt. CID continues to issue prompts after processing commands until you enter the command to resume execution of your program or terminate the session.

COMMAND FORMATS

CID commands are of two types: standard and special FORTRAN commands. Standard CID commands consist of a command name followed by a list of parameters separated by commas. For example, the command to display the first five words of the array A in octal format is:

```
DISPLAY,A,O,5
```

In standard CID commands, certain parameters are optional; if an optional parameter is omitted, but a subsequent parameter appears, the omitted parameter is indicated by two successive commas, as in the command:

```
DISPLAY,A,,5
```

The second command type is limited to FORTRAN programs compiled with the DB option. The form of these commands is identical to corresponding FORTRAN statements. An example of such a command is:

```
PRINT*,"VALUE IS",X
```

Commands are normally entered one per line, but more than one command can be included on a line if separated by semicolons, as in the following example:

```
?SET,TRAP,LINE,*; SET,BREAKPOINT,L.10; GO
```

Blanks are ignored and can be inserted anywhere in a line. A command cannot span more than one line.

SHORTHAND NOTATION

Most standard CID commands have a shorthand form that permits you to omit the comma separator and to substitute abbreviations for the command name and certain parameters. For example, the command:

```
SET,TRAP,LINE,*
```

can be expressed as:

```
ST L *
```

The shorthand notation provides a more convenient method of specifying commands, and you are encouraged to use this form as you become more familiar with CID. However, for purposes of clarity and consistency, only the full command forms are used in this manual. The short command forms are listed in appendix E.

SOME ESSENTIAL COMMANDS

The following paragraphs describe three CID commands that enable you to conduct simple debug sessions. These are the GO command, the QUIT command, and the PRINT command. These commands are discussed in greater detail in section 3.

GO COMMAND

The command to initiate or resume program execution is:

```
GO
```

This command causes execution to begin at the location where it was suspended.

Once execution of your program has been suspended, any number of CID commands can be entered. Execution remains suspended until GO is entered.

QUIT COMMAND

The command to terminate a debug session is:

```
QUIT
```

In response to the QUIT command CID displays the following message under NOS/BE and NOS batch subsystem:

```
DEBUG TERMINATED
```

Under NOS FTNTS subsystem:

```
SRU        n.nnn UNTS

RUN COMPLETE.
```

The QUIT command causes an exit from the current session and a return to system command mode. Files accessed by the FORTRAN program are closed. Note, however, that debug mode remains on until DEBUG(OFF) is specified.

Traps, breakpoints, and other alterations to the object program exist only for the duration of the debug session. When the session is terminated, any changes made to the program are lost and the program reverts to its compiled version. The object program can be rewound and executed normally or again under control of CID.

PRINT COMMAND

CID provides several commands for displaying the values of program variables. The simplest of these is the command:

```
PRINT*,list
```

where list is a list of program variables.

This command lists the values of the specified program variables. Values are formatted according to type declared, implicitly or explicitly, in the source program (integer, real, logical, or complex).

SAMPLE DEBUG SESSION

The preceding commands are now used to conduct a simple debug session.

A FORTRAN program and debug session log are illustrated in figure 2-2. The program defines two variables and performs a simple computation. The program is executed under CID control. Since no provision is made for suspending execution, the program runs to completion. CID then displays the message:

```
*T#17,END IN L.5
?
```

This is a trap message, explained in section 3. CID automatically sets a trap to gain control on program termination. The PRINT command prints the values of the specified program variables, and the QUIT command terminates the session.

PROGRAM LISTINGS

To use CID effectively you should have a program listing to which you can refer. The particular listing you should use depends on whether you are debugging under NOS in the batch subsystem, under NOS in the FTNTS subsystem, or NOS/BE INTERCOM.

CID references sequence numbers located at the beginning of FORTRAN source statements. Programs compiled under NOS with the RUN command have line numbers inserted in columns 1 through 5 by the NOS line editor. In this case you should use a listing produced by the LIST or LNH command. Under NOS/BE, EDITOR inserts sequence numbers in columns 76 through 80 and CID uses the numbers inserted at the beginning of each statement by the compiler. Thus, when using the RUN,FTN command, you cannot use the listing produced by LIST,ALL which shows the EDITOR-produced sequence numbers at the beginning of each statement. In this case, you can use a listing produced by LIST,ALL,SEQ and write in line numbers yourself according to the way the compiler does it. The FORTRAN compiler numbers statements in increments of 1 starting at 1. Line numbering starts anew at the beginning of each subprogram.

When compiling with the FTN statement, you can obtain a compiler output listing by specifying the L parameter on the FTN control statement and printing the resulting output file. Examples of source line references are illustrated under Program Address Notation.

PROGRAM ADDRESS NOTATION

CID provides notation for specifying single addresses within a program as well as ranges of successive addresses. The address notations most commonly used by FORTRAN programmers are summarized in tables 2-1 and 2-2. These notations are also used by CID in informative messages and other types of output.

If a program contains multiple subprograms, a qualified address form can be used to identify an address in a particular subprogram. Unqualified addresses are assumed to belong to a program unit called the home program.

Not all address formats are valid for all CID commands. Valid formats for each command are noted in section 3.

Program COMPC:

```
1               PROGRAM COMPC      74/74   TS ID


                                PROGRAM COMPC
                                A=3.0
                                B=2.0
                                C=(A+B)/(A-B)
              5                 STOP
                                END
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?go                                          <- Initiate program execution.

*T #17, END IN L.5                           <- Program runs to completion; END trap gives control to user.
?
     STOP
       .000 CP SECONDS EXECUTION TIME        <- Note: These FORTRAN messages appear on program
                                                termination, usually immediately after the Debug
                                                input prompt.
print*,a,b,c                                 <- Print the values of A, B, and C.

 3. 2. 5.
?quit                                        <- Terminate session.

  DEBUG TERMINATED
```

Figure 2-2. Sample Program and Debug Session

TABLE 2-1. ADDRESS NOTATION

Specification	Description
var	Simple or subscripted variable in home program.
L.n	Line n in home program.
S.n	Statement with label n in home program.
P.prog_var	Simple or subscripted variable in program unit prog.
P.prog_L.n	Source line n in program unit prog.
P.prog_S.n	Source statement with label n in program unit prog.
C.n	n+1st location of unlabeled common; $n \geq 0$.
C.blk_n	n+1st location of common block blk; $n \geq 0$.
XC.blk_n	n+1st location of ECS/LCM common block blk; $n \geq 0$.

HOME PROGRAM

FORTRAN programs consist of a main program and, optionally, one or more subprograms. Variable names within a program are local to the program unit in which they are defined; that is, variable names are known only within the program in which they are used. This concept is illustrated in figure 2-3. In this example, the variable A is defined twice: once in the main program and once in the subroutine. However, execution of the statement A=1.0 in the subroutine does not alter the contents of the variable A in the main program; the value printed for A is always 1.0. Although two variables have the same name, each is local to the program unit in which it is defined.

```
                PROGRAM MAIN(OUTPUT)
                A=1.0
                CALL SUBA
                PRINT*," A= ",A
                STOP
                END
      C
                SUBROUTINE SUBA
                A=2.0
                RETURN
                END
```

Figure 2-3. Program and Subroutine Illustrating Local Variables

TABLE 2-2. ADDRESS RANGE SPECIFICATION

Specification	Description
P.prog	Range of addresses occupied by program unit prog.
C.	Range of addresses occupied by unlabeled common.
C.blk	Range of addresses occupied by common block blk.
XC.blk	Range of addresses occupied by ECS/LCM common block blk.
var...var+n	Location of var and n succeeding locations.
L.n_1...L.n_2	Sources lines n_1 through n_2, inclusive, of home program.
S.n_1...S.n_2	Source statement labeled n_1, n_2, and all statements in between, in home program.
P.prog_var...P.prog_var+n	Location of var and n succeeding locations, in program unit prog.
P.prog_L.n_1...P.prog_L.n_2	Source lines n_1 through n_2, inclusive, in program unit prog.
P.prog_S.n_1...P.prog_S.n_2	Source statements labeled n_1, n_2, and all statements in between, in program unit prog.
C._n_1...C._n_2	Locations n_1+1 through n_2+1 of unlabeled common.
C.blk_n_1...C.blk_n_2	Locations n_1+1 through n_2+1 of common block blk.
XC.blk_n_1...XC.blk_n_2	Locations n_1+1 through n_2+1 of ECS/LCM common block blk.

The same concept of locality applies to CID. When a program containing several subroutines is executed under CID control, execution can be suspended in the main program or in any of the subprograms. By default, the home program is the program unit in control at the time of suspension. Addresses specified in most CID commands are local to the home program, unless appropriate qualifiers are specified. Any program unit can be designated at the home program by the SET,HOME command, as described under Referencing Addresses Outside the Home Program.

The home program concept is illustrated by the debug session in figure 2-4, produced by executing the program in figure 2-3 in debug mode. Breakpoints are set to suspend execution in the main program after the call to SUBA, and in SUBA itself. When execution is suspended in SUBA, SUBA is the home program and the PRINT command shows 2.0 as the value of A; when execution is suspended in the main program, the value of A is 1.0.

SPECIFYING A SINGLE ADDRESS

Most CID commands require the specification of at least one program address to identify a program variable or statement location. For example, to display the contents of a variable you must specify the address of the variable; to set a breakpoint at a location within a program you must specify the location address.

CID allows you to reference program addresses symbolically by specifying:

- Variable name
- Statement label
- Source line number

A program location can also be referenced by specifying its relative octal address; however, this method is seldom used by FORTRAN programmers. The following paragraphs describe the available methods of address specification.

Variable Name

Variables within a home program can be referenced simply by specifying the variable name, as in the following examples:

PRINT*,X,Y
Display the value of the variables X and Y.

DISPLAY,A
Display the value of the variable A.

X=Y+Z
Calculate the value of Y+Z and store it in X.

Source Line Number

A statement within a home program can be referenced by specifying the line number of the statement. The format of a line number specification is:

L.n

where n is the line number. The line numbers referenced by CID depend on how the program was compiled.

- If the program was compiled under NOS in the FTNTS subsystem, CID uses line numbers generated by the line editor.

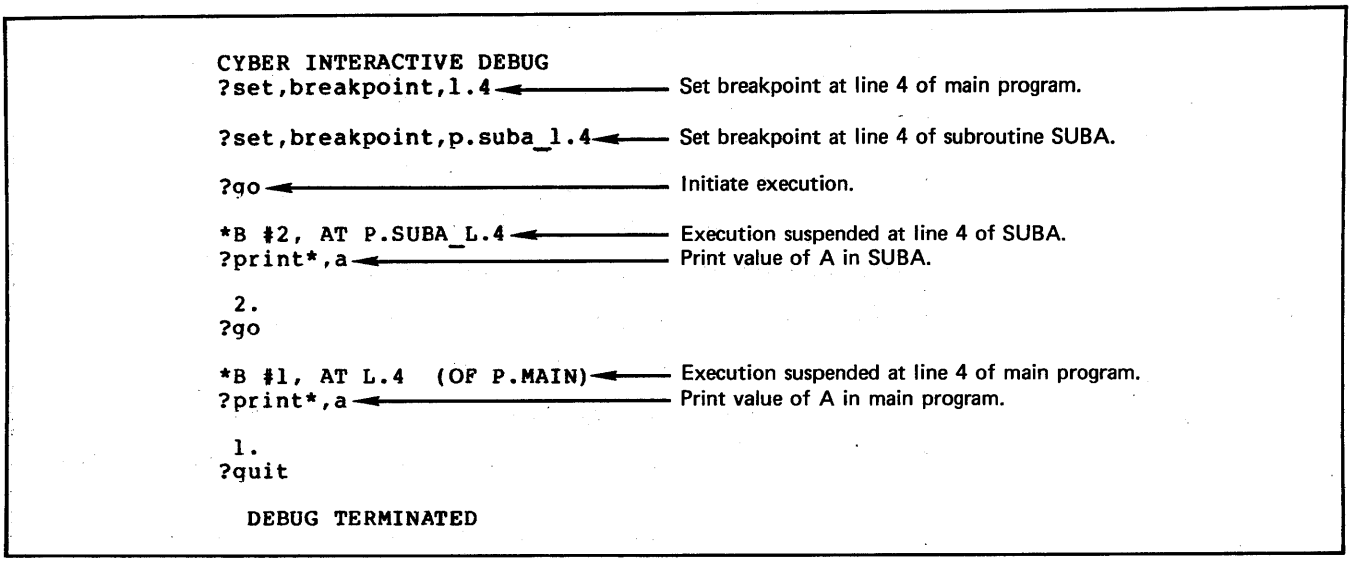

Figure 2-4. Debug Session Illustrating Local Variables

- If the program was compiled with the FTN statement or by the RUN,FTN command under NOS/BE, CID uses line numbers generated by the FORTRAN compiler.

Example 1 of figure 2-5 illustrates a program to be debugged under NOS. The listing was produced by entering the LIST command while in the FTNTS subsystem under NOS. By default, lines are numbered consecutively in increments of 10 starting at 100. These values can be changed with the RESEQ command. Leading zeros can be omitted when referencing a source line. For example, the command:

SET,BREAKPOINT,L.200

sets a breakpoint at the statement IF(R(I).GT.9999.0) GO TO 14, assuming that subroutine COMP is the home program.

Example 2 of figure 2-5 illustrates a program to be debugged under NOS/BE. The listing was produced by the FORTRAN compiler. Lines are numbered consecutively in increments of 1 starting at 1; numbers appear on the listing at every fifth line. In a program containing multiple program units, line numbering starts anew after the end of each subprogram. In this example, the command:

SET,BREAKPOINT,L.4

refers either to the statement N=10 in the main program, or to the statement IF(R(I).GT.9999.0) GO TO 14 in the subroutine, depending on which is the home program.

In most command formats the L.n notation can only be used to reference executable statements.

Statement Label

Program statements can be referenced by specifying a label assigned to the statement in the source program. The format of a statement label specificaton is:

S.n

where n is the statement label. Only executable statements can be referenced in this manner.

Source statement specifications are most often used when setting or referencing breakpoints at particular locations. For example, referring to the program in figure 2-5, the command:

SET,BREAKPOINT,S.14

sets a breakpoint at the statement 14 CONTINUE, if subroutine COMP is the home program.

SPECIFYING AN ADDRESS RANGE

Some command formats require the specification of a range of successive addresses. An address range is generally specified under the following circumstances:

- To display or alter the contents of an array or common block
- To limit the scope of a CID command to a particular program unit or portion of a program unit

The notation used for specifying address ranges is summarized in table 2-2. Only those formats useful to FORTRAN programmers are discussed here. Refer to the CYBER Interactive Debug reference manual for additional information on address range specification.

Program Unit Specification

The range of addresses occupied by a FORTRAN program unit (main program or executable subprogram) is denoted by the notation:

P.prog

where prog is a program unit name.

Example 1. NOS Time-Sharing System Listing:

```
00100 PROGRAM INIT (INPUT,OUTPUT)
00110 DIMENSION R(10),S(10),T(10),X(10),Y(10),Z(10)
00120 DATA R/10*1.6/,S/10*-3.5/,T/10*4.1/
00130 N=10
00140 CALL COMP (N,R,S,T,X,Y,Z)
00150 STOP
00160 END
00170 SUBROUTINE COMP (N,R,S,T,X,Y,Z)
00180 DIMENSION R(N),S(N),T(N),X(N),Y(N),Z(N)
00185 IF (R(I).GT.9999.0) GO TO 14
00190 DO 14 I=1,N
00200 X(I)=R(I)+S(I)
00210 Y(I)=R(I)+T(I)
00220 Z(I)=S(I)+T(I)
00230 14 CONTINUE
00240 RETURN
00250 END
```

Example 2. Compiler Output Listing:

```
1             PROGRAM INIT       73/74   TS ID

                                 PROGRAM INIT (INPUT,OUTPUT)
                                 DIMENSION R(10),S(10),T(10),X(10),Y(10),Z(10)
                                 DATA R/10*1.6/,S/10*-3.5/,T/10*4.1/
                                 N=10
          5                      CALL COMP (N,R,S,T,X,Y,Z)
                                 STOP
                                 END

1          SUBROUTINE COMP       73/74   TS ID

                                 SUBROUTINE COMP (N,R,S,T,X,Y,Z)
                                 DIMENSION R(N),S(N),T(N),X(N),Y(N),Z(N)
                                 DO 14 I=1,N
                                 IF (R(I).GT.9999.0) GO TO 14
          5                      X(I)=R(I)+S(I)
                                 Y(I)=R(I)+T(I)
                                 Z(I)=T(I)+S(I)
                        14       CONTINUE
                                 RETURN
         10                      END
```

Figure 2-5. Program Listings for Use With CID

As will be shown later in this section, it is often necessary to specify a program unit name as part of a CID command when you are debugging a program containing several subprograms. For example, the command:

SET,TRAP,JUMP,P.SETB

sets a JUMP trap to occur at all branches within the program unit SETB.

Common Block Specification

The locations occupied by a common block are designated by the following notation:

Common Block Type	Notation
Labelled common block (central memory)	C.name
Labelled common block (ECS/LCM)	XC.name
Unlabelled common block	C.

This notation is used to reference a common block in its entirety. For example, the command:

SET,TRAP,STORE,C.BLKA

sets a STORE trap for each location of common block BLKA.

To denote a particular location within a common block, a decimal integer offset can be specified as part of the common block specification. For example,

C.BCOM_2

denotes the third word of common block BCOM. The first word of a common block is designated by an offset of 0. An underscore character separates the block name from the offset. (The underscore character is displayed as an arrow (←) on some ASCII terminals.)

Ellipsis Notation

Certain commands allow you to specify an address range by including an ellipsis between the first and last address in the range as follows:

address1...address2

where address1 and address2 are source line specifications (L.n) or statement label specifications (S.n). If the S.n notation is used, the first label must precede the second label in the source program.

The ellipsis notation can also have the form:

var...var+n

where var is a variable name and n is a decimal integer offset. This form specifies the n locations var through var+n, and can be used to indicate an array, as described under Array Specification.

Examples:

SET,TRAP,LINE,L.4...L.20
Set a line trap at source lines 4 through 20.

DISPLAY,X...X+4
Display the values stored at four successive locations starting at X.

SET,TRAP,JUMP,S.10...S.50
Set a JUMPTRAP at the statement labeled 10 and succeeding statements, through the statement labeled 50.

Array Specification

The notation used to specify an array to CID depends on the command. For example, to display the contents of an array with the PRINT command, it is necessary to specify only the first location of the array. Thus, if a program contains the statement DIMENSION A(10) then the command:

PRINT*,A

displays the contents of each of the 10 words of A.

For other CID commands, however, specifying the array name designates only the first location of the array. For these commands, the ellipsis notation is used to designate an array, as in the following examples:

DISPLAY,A...A+9
Display the contents of A(1) through A(10).

DISPLAY,A
Display the contents of A(1).

SET,TRAP,STORE,A+4...A+6
Set a store trap for array elements A(5), A(6), and A(7).

In ellipsis notation, note that A corresponds to A(1), A+1 corresponds to A(2), and so forth.

REFERENCING ADDRESSES OUTSIDE THE HOME PROGRAM

In some cases, you might wish to reference a location in a program unit other than the home program. For example, when execution is suspended in the main program, you might want to set a breakpoint or display a value local to another subprogram. To accomplish this you can do either of the following:

- Use CID commands which allow an address specification to be qualified by a program unit name.
- Designate a new home program with the SET,HOME command.

Address Qualifiers

Certain CID commands allow you to use address qualification notation to specify an address in a program other than the home program. An address qualifier has the form:

P.prog_loc

where prog is the name of a program unit and loc is an address within the program unit; loc can have one of the following forms:

L.n Source line number

S.n Statement label

Simple or subscripted variable name

The character separating prog and loc is an underscore character. (The underscore character is printed as an arrow (←) on some ASCII terminals.)

This notation is valid for the SET, DISPLAY, LIST, CLEAR, ENTER, and MOVE commands; it is not valid for the PRINT, IF, and assignment commands.

The following examples refer to the program in figure 2-3:

DISPLAY,P.MAIN_A
Display the contents of A as defined in MAIN.

DISPLAY,P.SUBA_A
Display the contents of A as defined in subroutine SUBA.

SET,BREAKPOINT,P.SUBA_L.190
Set a breakpoint at line 190 of subroutine SUBA.

Address qualifiers can be used in conjunction with ellipsis notation to specify a range of addresses within a particular program unit, as in the command:

SET,TRAP,LINE,P.GETY_L.10...P.GETY_L.20

This command establishes a LINE trap at lines 10 through 20 of program unit GETY.

Additional examples of the use of address qualifiers are presented in the discussions of the individual commands.

SET,HOME Command

As an alternative to the address qualification notation, or in cases where this notation is invalid, you can specify addresses outside the default home program by first issuing the command:

SET,HOME,P.prog

where prog is a program unit name. This command changes the home program for purposes of address specification. Any unqualified addresses specified after entering the SET,HOME command belong to prog. It is important to note that the SET,HOME command does not alter the location where execution resumes when you issue GO or EXECUTE; execution always resumes either at the location where it was suspended or at the address specified in the GO or EXECUTE command regardless of SET,HOME specification. In addition, when execution is resumed, a previous SET,HOME specification is lost and the home program reverts to the one currently executing.

The debug session in figure 2-6, produced by executing the program in figure 2-3 in debug mode, illustrates the SET,HOME command. Note that on program termination, the home program is once again the main program; to print A in SUBX, a SET,HOME must be issued.

CONNECTED FILES

Programs using connected files can be executed under CID control. When using connected files it is helpful to code your program in such a way as to differentiate between a program request for input and a CID request for input. Likewise, you should have some method of distinguishing program output from CID output. This is particularly important when running under NOS since the system automatically inserts a ? prompt, identical to the CID prompt, at the beginning of a line to indicate a program request for user input.

An example of connected file usage under CID control is illustrated in figure 2-7 (NOS) and figure 2-8 (NOS/BE). Program ATR reads the coordinates of the vertices of a triangle and calculates the area of the triangle. Files INPUT and OUTPUT are used so that input and output can be performed through the terminal. Immediately before the READ is executed, a WRITE statement displays two asterisks (**) to indicate that the program is waiting for user input. Input data is then entered on the same line as the asterisks. After the final calculation, a WRITE statement displays a message and the calculated area.

The NOS session is slightly more confusing because of the system-issued ? prompt. The two asterisks, however, identify the subsequent ? as being issued by NOS and not by CID.

```
CYBER INTERACTIVE DEBUG
?go

 A= 1.
*T #17, END IN L.5 <------------------------ Program terminates.
?
     STOP
        .050 CP SECONDS EXECUTION TIME
print*,a <---------------------------------- Print value of A in home program (MAIN).

 1.
?set,home,p.suba <-------------------------- Designate SUBA as home program.

?print*,a <--------------------------------- Print value of A in home program (SUBA).

 2.
?quit

  DEBUG TERMINATED
```

Figure 2-6. Debug Session Illustrating SET,HOME Command

```
00100 PROGRAM ATR (INPUT,OUTPUT)
00110 10 WRITE 100
00120 100 FORMAT(" ** ")
00130 READ 150, X1,Y1,X2,Y2,X3,Y3
00140 150 FORMAT(6F6.2)
00150 IF(X1.EQ.9999.0) STOP
00160 S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
00170 S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
00180 S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
00190 T=(S1+S2+S3)/2.0
00200 A=SQRT(T*(T-S1)*(T-S2)*(T-S3))
00210 WRITE 200, A
00220 200 FORMAT(" AREA IS ",F8.2)
00230 GO TO 10
00240 END


 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.200          <-- Set breakpoint at line 200.
? go                            <-- Initiate execution.
 **                             <-- Program writes asterisks.
?  1.0   2.4   -5.1  0.4   -2.2  0.9   <-- System issues input prompt. User enters input data.
 *B #1, AT L.200                <-- Execution suspended at line 200.
? print*,s1,s2,s3               <-- Display intermediate values.
  6.4195015382816 3.5341194094145 2.9427877939124
? go                            <-- Resume execution.
 AREA IS      1.37              <-- Program writes message and value.
 **
? 9999.0                        <-- User enters 9999.0 to indicate end of input.
 *T #17, END IN L.150
? quit                          <-- Terminate session.

SRU       5.719 UNTS.

 RUN COMPLETE.
```

Figure 2-7. Program ATR and Debug Session Illustrating Connected Files (NOS)

```
1            PROGRAM ATR        73/74   TS ID


                        PROGRAM ATR(INPUT,OUTPUT)
                   10   WRITE 100
                   100  FORMAT(" ** ")
                        READ 150, X1,Y1,X2,Y2,X3,Y3
         5         150  FORMAT(6F6.2)
                        IF(X1.EQ.9999.0) STOP
                        S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
                        S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
                        S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
        10              T=(S1+S2+S3)/2.0
                        A=SQRT(T*(T-S1)*(T-S2)*(T-S3))
                        WRITE 200, A
                   200  FORMAT(" AREA IS ",F6.2)
                        GO TO 10
        15              END


CYBER INTERACTIVE DEBUG
?set,breakpoint,l.11 <-- Set breakpoint at line 11.

?go <-- Initiate execution.
    <-- Program is ready for input.
**1.0   2.4   -5.1  0.4   -2.2  0.9 <-- User enters input data.

*B #1, AT L.11 <-- Execution suspended at line 11.
?print*,s1,s2,s3 <-- Display intermediate values.

 6.41950153828l6 3.5341194094145 2.9427877939124
?go <-- Resume execution.

AREA IS   1.37 <-- Program writes message and value.
**9999.0 <-- User enters 9999.0 to indicate end of input.

*T #17, END IN L.6
?quit <-- Terminate session.
     STOP
       .249 CP SECONDS EXECUTION TIME


  DEBUG TERMINATED
```

Figure 2-8. Program ATR and Debug Session Illustrating Connected Files (NOS/BE)

Once you have compiled your FORTRAN program in debug mode and initiated a debug session you are ready to begin interactive debugging. Program execution under CID control involves an interaction between you and CID; you specify conditions for which program execution is to be suspended, and CID gains control when these conditions are satisfied and allows you to enter various CID commands to examine and alter the status of the program.

The preceding section presented some elementary commands that can be used to conduct a simple debug session. This section presents additional commands that will allow you to make more productive use of CID. The commands discussed in this section provide the capability to:

- Suspend program execution; commands are SET,BREAKPOINT and SET,TRAP.
- Display the current contents of program variables and arrays at the terminal while execution is suspended; commands are PRINT, DISPLAY, and LIST,VALUES.
- Alter the contents of variables and arrays; commands are MOVE and assignment.

TRAPS AND BREAKPOINTS

When conducting a debug session, you must initially provide for gaining interactive control at some point or points within your program. CID provides two methods of doing this: traps and breakpoints.

A trap is a CID mechanism that detects the occurrence of a specified condition during program execution, suspends execution at that point, and transfers control to CID. A breakpoint is a location within the user program where execution is to be suspended.

In a typical debug session, traps or breakpoints are established prior to initiating execution of the program. When a trap condition occurs or a breakpoint is detected during execution, CID receives control and, in turn, gives you the opportunity to enter CID commands.

In most debugging situations, you should probably use breakpoints, rather than traps, to suspend execution. Traps can be useful in certain cases, but their use often requires you to be familiar with COMPASS instructions. In addition, certain traps greatly increase execution time. Breakpoints allow you to suspend execution at any executable statement in your program and can, in most cases, be substituted for traps.

Traps and breakpoints exist only for the duration of a debug session. Once a session is terminated, all traps and breakpoints set during a session cease to exist. An object program is not permanently altered by any traps or breakpoints established during a session.

SUSPENDING PROGRAM EXECUTION WITH BREAKPOINTS

A breakpoint is a mechanism established at a specified location within a program which, when the location is reached during program execution, suspends execution and gives control to CID. CID then allows you to issue commands.

SET,BREAKPOINT COMMAND

The command to establish a breakpoint has the form:

```
SET,BREAKPOINT,loc
```

where loc is a source line number (L.n) or a statement label specification (S.n). To set a breakpoint at a location not in the home program you can either specify a qualified address form (P.name L.n or P.name S.n) or designate a new home program with the SET,HOME command. Following are two examples of the SET,BREAKPOINT command:

```
SET,BREAKPOINT,L.14
SET,BREAKPOINT,P.MXY_L.25
```

The first command sets a breakpoint at line 14 of the home program. The second command sets a breakpoint at line 25 of program unit MXY.

Breakpoints can be established at any time in the debug session when execution is suspended and CID has issued a ? prompt.

A breakpoint can be established at any executable statement. Only one breakpoint can be set at a single statement; however, breakpoints can be set at locations where traps occur and all are recognized. Breakpoints are always recognized first.

Establishing a breakpoint at a specified location does not alter execution of the statement at that location. When a breakpoint is encountered during execution, CID gains control before the statement is executed. When execution is resumed, execution begins with the statement at the breakpoint location.

When a breakpoint is encountered, CID receives control and issues the following message:

```
*B #n AT loc
```

where n is a breakpoint number assigned by CID and loc is the statement (S.n or L.n) where the breakpoint was set. Breakpoints are assigned consecutive numbers in the order they are established, starting with 1. You can refer to breakpoints by number in the CLEAR command (described later in this section), LIST command (section 4), and SAVE command (section 5).

CID provides another form of the SET,BREAKPOINT command that is extremely useful for debugging DO loops and other sections of code that are executed frequently. The form of this command is:

SET,BREAKPOINT,loc,first,last,step

where first, last, and step are frequency parameters. This command sets a breakpoint that suspends execution every stepth time the breakpoint is reached, beginning with the first time and not after the last time. For example, the command:

SET,BREAKPOINT,L.50,10,100,5

sets a breakpoint at the statement labeled 50 which is recognized on the 10th time the statement is reached and every 5th time thereafter, up through the 100th time.

As an example of the use of the frequency parameters, consider the following loop:

```
      DO 8 I=1,1000
      X=X+FX/DX
8     CONTINUE
```

To examine the progress of the iteration X=X+FX/DX, you could set a breakpoint at statement 8, specifying frequency parameters to suspend execution at an interval rather than on each pass through the loop. For example,

SET,BREAKPOINT,S.8,1,1000,100

suspends execution on every 100th pass through the loop, starting with the first pass.

To illustrate the SET,BREAKPOINT command the program shown in figure 3-1 is executed under CID control. The debug session is shown in figure 3-2. The program consists of a main program and a subroutine called AREA. The main program reads data records, with each record containing the coordinates of the vertices of a triangle, and calls AREA. AREA calculates the area of the triangle. The purpose of the debug session is to suspend execution at the beginning of the subroutine to examine the input, and after the final calculation to examine intermediate values and final results. To accomplish this, breakpoints are set at lines 2 and 6. When execution is suspended at line 6, a breakpoint is set at line 7. The PRINT command displays the desired information. The QUIT command terminates the session after the first pass through AREA.

REMOVING BREAKPOINTS

Breakpoints can be removed during a debug session with the CLEAR,BREAKPOINT command. This command has the following forms:

CLEAR,BREAKPOINT
CLEAR,BREAKPOINT,list

Main Program and Subroutine:

```
1               PROGRAM RDTR        74/74   TS ID

                                    PROGRAM RDTR(TRFILE,TAPE2=TRFILE)
                                    REWIND 2
                              10    READ (2,*) X1,Y1,X2,Y2,X3,Y3
                                    IF(EOF(2).NE.0) GO TO 12
                    5               CALL AREA(X1,Y1,X2,Y2,X3,Y3,A)
                                    GO TO 10
                              12    STOP
                                    END

1             SUBROUTINE AREA       74/74   TS ID

                                    SUBROUTINE AREA(X1,Y1,X2,Y2,X3,Y3,A)
                                    S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
                                    S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
                                    S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
                    5               T=(S1+S2+S3)/2.0
                                    A=SQRT(T*(T-S1)*(T-S2)*(T-S3))
                                    RETURN
                                    END
```

Input Data:

```
0.0 1.0 0.5 2.0 -1.0 1.2
6.1 2.0 0.1 -4.0 3.2 7.0
0.2 -2.9 -1.3 8.0 5.6 2.8
```

Figure 3-1. Subroutine AREA, Main Program, and Input File

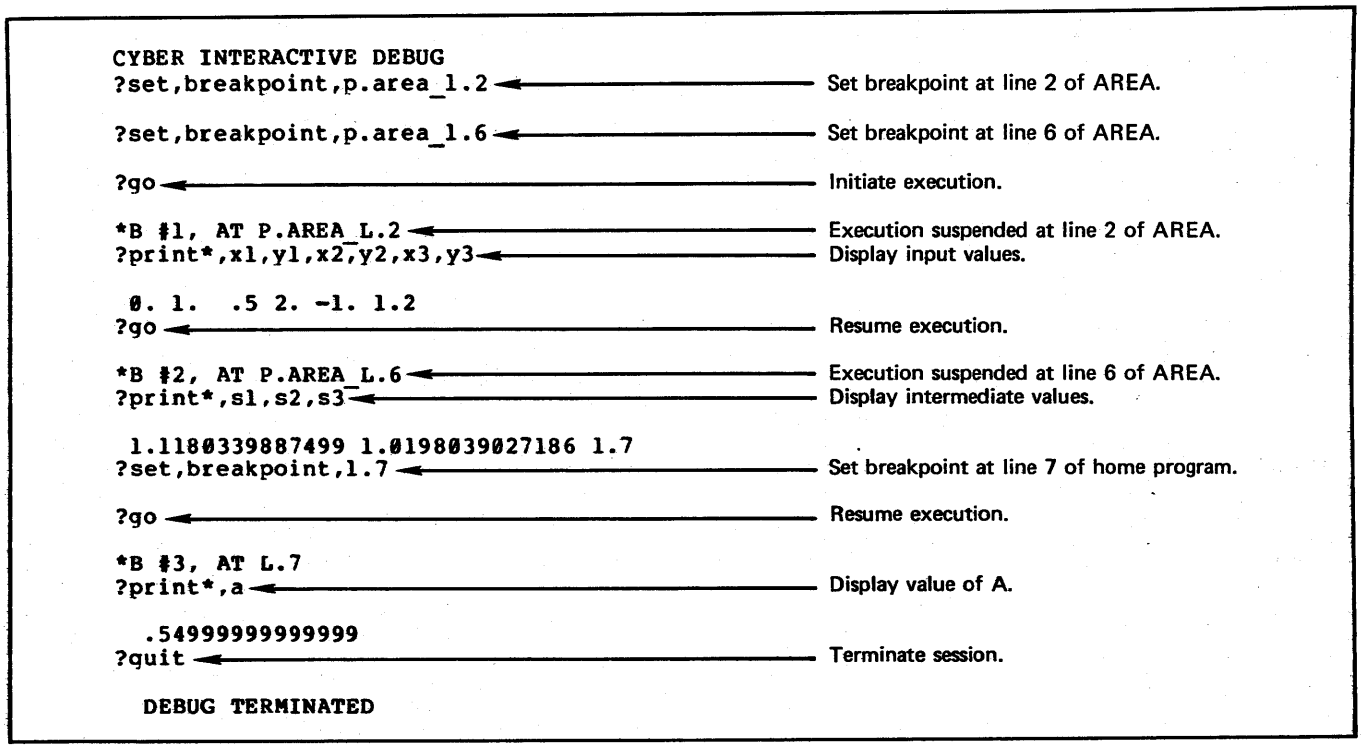

Figure 3-2. Debug Session Illustrating SET,BREAKPOINT Command

where list is a list of breakpoint locations, separated by commas. The first form clears all breakpoints. The second form clears the specified breakpoints. The list parameter can have any of the forms shown in table 3-1.

The CLEAR,BREAKPOINT command should be used for removing breakpoints that are no longer needed in a debug session in order to eliminate unnecessary and time-consuming suspensions of execution.

Following are examples of the CLEAR,BREAKPOINT command:

CLEAR,BREAKPOINT,L.14,L.20,P.SUB3_S.5
Remove the breakpoints from lines 14 and 20 of the home program and from the statement labeled 5 in program unit SUB3.

CLEAR,BREAKPOINT,P.READXY,P.ADDR
Remove all breakpoints from program units READXY and ADDR.

CLEAR,BREAKPOINT,P.MTX_L.4...P.MTX_L.100
Remove all breakpoints from lines 4 through 100 of program unit MTX.

CLEAR,BREAKPOINT,#3,#5,#6
Remove breakpoints identified by numbers 3, 5, and 6.

TABLE 3-1. OPTIONS FOR CLEAR,BREAKPOINT COMMAND

List Parameter	Explanation
$loc_1,loc_2,...$	Clears the breakpoints from the specified locations; locn can have any of the following forms: L.n S.n P.name_L.n P.name_S.n
$loc_1...loc_2$	Clears all breakpoints from the specified inclusive range; locn can have any of the following forms: L.n S.n P.name_L.n P.name_S.n
$P.name_1,P,name_2,...$	Clears all breakpoints from the specified program units.
$\#n_1,\#n_2,...$	Removes the breakpoints identified by the specified numbers.

SUSPENDING PROGRAM EXECUTION WITH TRAPS

A trap is a special condition within a program which causes control to automatically transfer to CID whenever that condition is detected during execution of the program.

The most useful traps to the FORTRAN programmer are the LINE and STORE traps and the default END and ABORT traps. These traps are discussed first. The JUMP and RJ traps are less useful to the average programmer and are discussed at the end of this section. The OVERLAY trap is used only in programs with overlays; this trap is discussed in section 6.

Conditions for which traps can be established are listed in table 3-2. Only those traps considered most useful to FORTRAN programmers are discussed here. Refer to the CID reference manual for information on other traps.

When a trap condition is detected, execution is suspended and CID gains control and issues a message identifying the trap, followed by a ? prompt for user input. The message gives information about the trap, including the trap type, number, and the address (L.n or S.n) of the location where the trap occurred. The trap number is a decimal integer assigned by CID. Traps are numbered sequentially in the order they are established. You can reference traps by number in the LIST, CLEAR, and SAVE commands. An example of a trap message is:

```
*T #3, RJ IN P.SBX_L.5
?
```

In this example, an RJ trap is detected in line 5 of program unit SBX; this trap was the third one established by the programmer.

In response to the ? prompt you can enter any CID command. Typically, you will use this opportunity to examine the values of program variables and make any desired changes to these values. Program execution can be resumed by entering a GO command.

A NOTE ON TRAPS

It is important to note that most of the CID traps occur when a particular machine instruction is detected. Some traps, such as the FETCH and STORE traps, occur after the instruction is executed; others, such as the JUMP trap, occur before the instruction is executed. Table 3-1 indicates whether a particular trap occurs before or after the instruction is executed.

Since a FORTRAN statement usually generates several machine instructions, confusion can arise as to the precise point in the execution of a statement at which the trap occurred. The trap message indicates only the number of the statement that was executing when the trap occurred. The point in the execution of a statement at which each trap type suspends execution is stated in the discussion of the trap type.

DEFAULT TRAPS

CID provides default traps that are set automatically at the beginning of a debug session. These traps allow you to gain control without actually establishing any traps or breakpoints. The default traps are the END, ABORT, and INTERRUPT traps.

The END and ABORT traps together transfer control to CID on any program termination. Thus, for the initial debug session, you can allow your program to terminate; by examining the status of the program at the point of termination, you can determine where traps or breakpoints should be set for subsequent sessions.

INTERRUPT Trap

An INTERRUPT trap occurs under the following circumstances:

- Program execution exceeds the allowed time limit.
- You issue a terminal interrupt.

TABLE 3-2. TRAP TYPES

Trap Type	Short Form	Condition	Established By	CID Gets Control
LINE	L	Beginning of an executable statement	User	Before the statement is executed
STORE	S	Store to memory	User	After the store
RJ		RETURN JUMP instruction (subprogram call or return)	User	Before the call or return is executed
JUMP	J	JUMP instruction	User	Before the jump is executed
FETCH	F	Fetch from memory	User	After the fetch
OVERLAY	OVL	Overlay load	User	After the overlay is loaded
INTERRUPT	INT	User interrupt or time limit	Default	After the interrupt
END	E	Normal program termination	Default	After termination
ABORT	A	Abnormal program termination	Default	After termination

If your program exceeds the maximum time limit allowed by the system, CID gains control and issues an informative message. Suggested user action when this occurs is discussed in section 1.

A terminal interrupt allows you to gain control at any time during a debug session. A terminal interrupt is issued by keying %A (NOS/BE), the BREAK key (NOS), or a) followed by a RETURN (NOS IAF). CID issues a trap message and input prompt, allowing you to enter CID commands.

When an executing program is interrupted, execution is suspended at the beginning of a FORTRAN statement.

The INTERRUPT trap can be used to terminate excessive output to the terminal, though it will cause the remaining output to be lost. It can also be used to interrupt a program that you believe to be looping excessively at some unknown location.

END Trap

The END trap suspends program execution on normal program termination. This trap always occurs when a program terminates normally, regardless of any CID commands that have been entered to set or clear traps.

The debug session in figure 2-1 illustrates the END trap. The program runs to completion and CID gains control and issues the message:

```
*T #17, END IN L.5
?
```

CID permanently assigns the number 17 to the END trap. In response to the ? prompt you can display program variables as they exist at the time of termination or you can terminate the session by entering QUIT. You cannot issue a GO or EXECUTE following an END trap.

ABORT Trap

The ABORT trap is extremely useful in that it allows you to gain interactive control on any abnormal termination of program execution. The status of program variables can be examined as they exist at the precise time of termination.

To illustrate how the ABORT trap works, an error that causes abnormal termination is introduced into the program shown in figure 2-1, and the program is executed under CID control. The source listing containing the error and the session log are shown in figure 3-3. The statement C=(A+B)/(A-B) results in a division by zero. The variable C is set to an infinite value (represented by the character R) and when C is used as an operand in the next statement, the program aborts with a mode 2 error. CID immediately gains control and issues the trap message indicating the trap type, number, location, and the error number. The user enters the LIST,VALUES command to display the contents of program variables. Note that in this case, the value of the infinite operand is represented by the character I. The QUIT command terminates the session.

The ABORT trap is permanently assigned the number 18 by CID.

Program COMPC:

```
1             PROGRAM COMPC      74/74   TS ID

                          PROGRAM COMPC
                          A=2.0
                          B=2.0
                          C=(A+B)/(A-B)
            5             D=D+1.0
                          STOP
                          END
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?go <-------------------------------------------- Initiate execution.

*T #18, ABORT CPU ERROR EXIT 04 IN L.5 ---------- ABORT trap occurs at line 5.
?print*,a,b,c,d <-------------------------------- Values of program variables.

 2. 2. R I <-------------------------------------- C is out of range and D is indefinite.
?quit <------------------------------------------- Terminate session.

  DEBUG TERMINATED
```

Figure 3-3. Program COMPC and Debug Session Illustrating ABORT Trap

USER-ESTABLISHED TRAPS

In addition to the default traps, CID provides traps that can be established and removed by the user.

SET,TRAP Command

The traps described in the following paragraphs are established with the SET,TRAP command. The format of this command is as follows:

SET,TRAP,type,scope

where type is one of the trap types listed in table 3-1 and scope is the address or range of addresses for which the trap is effective. The scope parameter has one of the forms listed in tables 2-1 and 2-2. Not all address notation forms are valid for all traps; the valid forms for each trap are stated in the discussion of the trap. For certain traps, an asterisk (*) can be specified for the scope parameter, in which case the trap is recognized throughout the entire program (unlimited scope). However, as will be demonstrated, it is not always practical to specify an unlimited scope. Normally, you will restrict the scope to a particular program unit.

Traps are typically established at the beginning of a debug session, although they can be established whenever execution is suspended and CID has issued a ? prompt. If a condition for which you have established a trap does not occur, the program executes normally.

Once you have established a trap, you can subsequently remove it with the CLEAR,TRAP command discussed later in this section. Although traps exist only for the duration of the debug session in which they are established, you can save trap definitions on a separate file for use in later sessions with the SAVE command discussed in section 5.

Following are some examples of the SET,TRAP command:

SET,TRAP,JUMP,P.BIRD
Suspend execution on all branches detected in program unit BIRD.

SET,TRAP,STORE,C.BLKA
Suspend execution whenever data is stored into a location in common block BLKA.

SET,TRAP,FETCH,P.SUBX_A
Suspend execution whenever data is fetched from the variable A in subroutine SUBX.

SET,TRAP,LINE,*
Suspend execution immediately before execution of each line in the user's program.

SET,TRAP,LINE,P.PROG1_L.5...P.PROG1_L.20
Suspends execution immediately before execution of each of the source lines 5 through 20 in program unit PROG1.

LINE Trap

The LINE trap gives control to CID immediately prior to execution of each executable FORTRAN statement within the specified range. The line trap allows you to examine and alter program status before each statement is executed.

For the scope parameter you can specify an asterisk, in which case execution is suspended after each FORTRAN statement in the user program, or you can limit the scope by specifying one of the address range formats listed in table 2-1. For example, the command:

SET,TRAP,LINE,L.5...L.12

sets a LINE trap of lines 5 through 12 of the home program.

To illustrate the LINE trap, the program in figure 3-4 is executed under CID control. The session log is shown in figure 3-5. The program consists of a main routine PROG1 and a subroutine SETB. The main routine contains two calls to SETB; SETB stores values into array B depending on the value of the variable K. The first CID command sets the LINE trap. The scope parameter specifies that the trap applies only to the main program. The trap occurs immediately before each executable statement. The command is entered after each subroutine call (execution suspended at lines 6 and 8). The GO command resumes execution after each suspension. Note that both the LINE and END traps occur at line 8, the last executable statement of the program. This illustrates that more than one trap can occur at the same location.

STORE and FETCH Traps

The STORE trap suspends execution whenever data is stored into the program locations specified by the scope parameter. The FETCH trap suspends execution whenever data is fetched from the specified locations. Execution is suspended immediately after the store or fetch is performed. The STORE trap is especially useful since it gives you control whenever the specified variable is modified.

A store instruction is generated whenever a variable name appears to the left of an equal sign, except as a subscript, or whenever data is stored as a result of an input statement. Any other reference to a variable, except in an argument list, generates a fetch instruction. Some examples of when a store or fetch occurs are as follows:

READ 10,X,Y	store X, store Y
IF(A.LT.B) GO TO 20	fetch A, fetch B
DO 10 I=1,N	store I, fetch N
A(I)=Z	fetch I, fetch Z, store A(I)

The scope parameter is generally specified as a variable name, as in the command:

SET,TRAP,STORE,X

which sets a STORE trap for the variable X.

To set a STORE or FETCH trap for an entire array, or part of an array, the ellipsis notation can be used.

```
1              PROGRAM MAIN        74/74   TS ID

                                   PROGRAM MAIN
                                   COMMON /BCOM/B(5)
                                   N=5
                                   K=1
                5                  CALL SETB(K,N)
                                   K=2
                                   CALL SETB(K,N)
                                   STOP
                                   END


1          SUBROUTINE SETB         74/74   TS ID

                                   SUBROUTINE SETB(K,N)
                                   COMMON /BCOM/B(5)
                                   IF(K.EQ.1) GO TO 8
                                   DO 6 I=1,N
                5          6       B(I)=-1.0
                                   RETURN
                           8       DO 12 I=1,N
                           12      B(I)=1.0
                                   RETURN
               10                  END
```

Figure 3-4. Subroutine SETB and Main Program

Specifying the array name for the scope parameter sets the trap only for the first word of the array. For example, if the statement DIMENSION A(10) appears in the FORTRAN program, then the command:

SET,TRAP,FETCH,A...A+9

sets a FETCH trap that suspends execution whenever data is fetched from any of the locations A(1) through A(10). The command:

SET,TRAP,FETCH,A

sets a FETCH trap that suspends execution whenever data is fetched from A(1).

Variable names can be qualified, as in the command:

SET,TRAP,FETCH,P.ADDB_X...P.ADDB_X+99

which sets a FETCH trap that suspends execution whenever data is fetched from any of the locations X(1) through X(100) in program unit ADDB.

When a common block name is specifed for the scope parameter, the trap is set for each location in the common block. For example, the command:

SET,TRAP,STORE,C.BLKX

sets a STORE trap that suspends execution whenever data is stored into any location in common block BLKX.

To set a trap for a single location within a common block, specify the block name and a decimal offset as in the command:

SET,TRAP,STORE,C.BLKX_0

which sets a STORE trap for the first location of BLKX.

It is important to note that the STORE and FETCH traps suspend execution immediately after the execution of the STORE or FETCH. This means that, in the case of the FETCH trap, the FORTRAN statement that caused the fetch does not execute to completion prior to suspension. For example, if the statement A=B+C appears in a source program and if a FETCH trap is set for the variable B, then execution is suspended after B is fetched and before the sum is calculated and stored into A. The function of the STORE and FETCH traps can be illustrated by examining the following object code generated by A=B+C:

```
SA5     B       fetch B
SA4     C       fetch C
SA1     CON1
FX0     X4+X5   sum operands
NX7     B0,X0
SA7     A       store A
```

By setting a STORE trap for A and FETCH traps for B and C, execution would suspend three times while executing these instructions.

A common bug in a FORTRAN program is a subroutine call with too few parameters. If a program contains many subroutine calls, a good trap to set is SET,TRAP,STORE,0, and SET,TRAP,FETCH,0. These traps will occur when a reference is made within the subroutine to the formal variable corresponding to the first missing parameter. An example of this type of error is illustrated at the end of the section.

An example of a debug session using the STORE and FETCH traps is illustrated in figure 3-6. The program in figure 3-4 is executed under CID control to produce this session log. Both the STORE and FETCH traps are set so that CID sets control whenever data is stored into or fetched from common block BCOM. Execution is subsequently suspended on each pass through the DO loop

```
CYBER INTERACTIVE DEBUG
?set,trap,line,p.main <-------- Set LINE trap in PROG1.

?go <-------- Initiate execution.

*T #1, LINE AT L.3
?print*,n

 ******************
?go

*T #1, LINE AT L.4
?go

*T #1, LINE AT L.5
?print*,n,k

 5 1
?go

*T #1, LINE AT L.6
?print*,b

 1. 1. 1. 1. 1.
?go

*T #1, LINE AT L.7
?print*,k

 2
?go

*T #1, LINE AT L.8
?print*,b

 -1. -1. -1. -1. -1.
?go

*T #17, END IN L.8 <-------- END trap occurs at line 8.
?quit <-------- Terminate session.
     STOP
       .782 CP SECONDS EXECUTION TIME


  DEBUG TERMINATED
```

LINE trap suspends execution at lines 3 through 8. After each suspension, selected values are displayed and execution is resumed. Undefined variables contain meaningless values.

Figure 3-5. Debug Session Illustrating LINE Trap

as the constant is stored into the five locations of the array B in common block BCOM. The FETCH trap does not occur since data is not fetched from locations in BCOM anywhere in the program.

JUMP Trap

The JUMP trap suspends program execution immediately prior to the execution of a jump instruction. The JUMP trap can be useful in tracing the flow of execution in programs that contain many branches. However, jump instructions are often generated by FORTRAN statements other than the normal branch statements, which can result in many unexpected suspensions of execution. In addition, the JUMP trap requires the much slower interpretive execution. For these reasons, it is usually better to substitute breakpoints for a JUMP trap.

Jump instructions are generated by the following statements:

- Unconditional branch (GO TO n)
- Arithmetic IF statement (IF(expr)n_1,n_2,n_3)
- Logical IF statement (IF(expr)GO TO n)
- DO loop repetition

The JUMP trap gives control to CID whenever any of the above program branches are detected. Note that for a logical IF statement, the trap occurs only if the logical expression is true. The JUMP trap does not detect function and subroutine calls.

The scope of a JUMP trap should always be restricted to a particular subprogram or main program. If * is specified

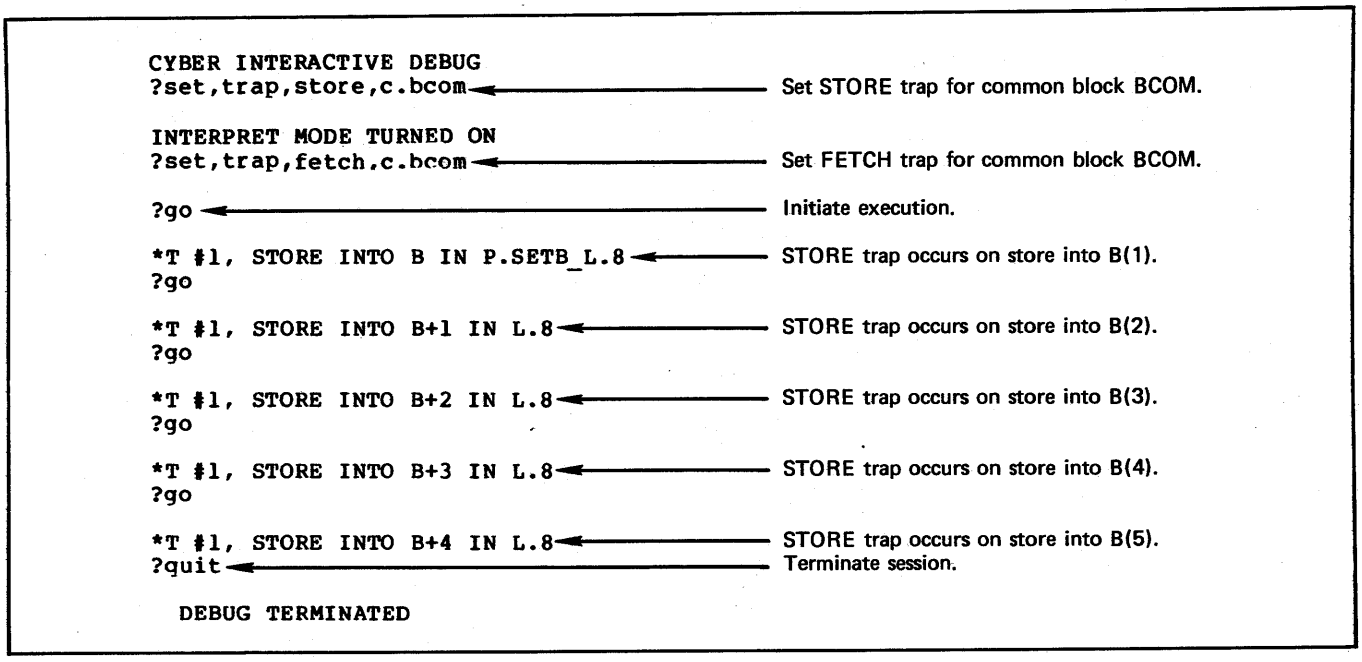

Figure 3-6. Debug Session Illustrating STORE and FETCH Traps

for the scope parameter, indicting unrestricted scope, then a trap is set at every jump instruction in the entire program including those in system and library routines. This results in many extraneous suspensions of execution. By specifying a particular routine name for the scope parameter, the range of addresses over which the trap is effective is limited to that routine. For example, the command:

```
SET,TRAP,JUMP,P.CAT
```

sets a jump trap that suspends execution at all jumps occurring in program unit CAT.

The scope parameter can also be specified as a range of source lines as in the command:

```
SET,TRAP,JUMP,P.DOG_L.4...P.DOG_L.20
```

which sets a JUMP trap that suspends execution at all jumps occurring in lines 4 through 20 in program unit DOG.

The JUMP trap can be useful for debugging DO loops since you get control on each pass through the loop. Execution is suspended at the end of the loop immediately after execution of the last statement but before the jump to the beginning of the loop. For example, if the following loop is located within the scope of a JUMP trap:

```
      DO 10 I=1,100
      XNEW=XNEW+X(I)
10    CONTINUE
```

then execution is suspended at the statement labeled 10 on each pass through the loop, allowing the current value of XNEW to be examined. If many passes are made through the loop, however, you should consider setting a breakpoint at the statement, instead of using a JUMP trap. Breakpoints provide the option of suspending execution on each pass though the loop or at specified intervals, and breakpoints do not require interpret mode.

A program and debug session illustrating the JUMP trap are shown in figure 3-7. Program TESTJ reads a single integer, tests the integer, and sets a variable A depending on the value of the integer. The SET,TRAP command establishes a JUMP trap in TESTJ. The scope of the trap is restricted to TESTJ so that execution will not be suspended on jumps occurring outside TESTJ. The trap occurs at lines 4, 11, and 15. After the first occurrence, the PRINT command displays the value of the input variable J. Note that the default END trap also occurs at line 15.

RJ Trap

The RJ trap gives control to CID whenever a return jump instruction is detected within the specified range. Return jumps are generated by subroutine calls, function references, and RETURN statements within a user program. Return jumps can also be generated by other FORTRAN statements, such as I/O statements, which cause the compiler to produce calls to external routines supplied by system libraries. An RJ trap causes execution to be suspended immediately before the return jump is executed.

The RJ trap can be useful for debugging programs that contain many subroutine calls. However, a FORTRAN program can contain many hidden return jump instructions, resulting in needless suspensions of execution. In addition, the RJ trap causes interpretive execution, a much slower mode of execution (described at the end of this section). In most cases, it is probably better to use breakpoints rather than an RJ trap.

The scope parameter is generally specified as one of the range formats shown in table 2-2. For example,

```
SET,TRAP,RJ,P.SUBR
```

sets an RJ trap that suspends execution at all return jump instructions occuring within program unit SUBR.

Program TESTJ:

```
1             PROGRAM TESTJ      74/74   TS ID


                          PROGRAM TESTJ(INPUT,OUTPUT)
                     5    WRITE*," ** "
                          READ*, J
                          IF(J.EQ.1) GO TO 10
          5               IF(J.EQ.2) GO TO 20
                          IF(J.EQ.3) GO TO 30
                          STOP
                     10   A=1.0
                          B=0.0
         10               GO TO 40
                     20   A=2.0
                          B=1.0
                          GO TO 40
                     30   A=3.0
         15               B=2.0
                     40   C=A+B
                          GO TO 5
                          END
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?set,trap,jump,p.testj                    <-- Set JUMP trap in main program.

INTERPRET MODE TURNED ON
?go                                       <-- Initiate execution.
                                          <-- Program writes input prompt.
" ** "3                                   <-- User enters input data.

*T #1, JUMP IN L.6                        <-- JUMP trap at line 6, IF statement.
?go

*T #1, JUMP AT L.17                       <-- JUMP trap at line 17, GOTO statement.
?print*,j,a,b,c

 3 3. 2. 5.
?go

" ** "4

*T #1, JUMP AT L.7                        <-- Hidden jump at line 7, STOP statement.
?go

*T #17, END IN L.7
?
     STOP
      9.452 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Figure 3-7. Program TESTJ and Debug Session Illustrating JUMP Trap

It is important to note that the RJ trap suspends execution at all return jumps occurring within the specified range. In addition to user-specified CALL statements, RETURN statements, and function references, execution is suspended at any FORTRAN statement that produces a call to a system library routine at compile time. FORTRAN statements producing return jumps include:

READ	BUFFEROUT
WRITE	PRINT
ENCODE	PUNCH
DECODE	REWIND
BUFFERIN	ENDFILE
	BACKSPACE

The RJ trap does not suspend execution at statement function references or at references to intrinsic functions since these do not generate return jump instructions.

If * is specified for the scope parameter, indicating unlimited scope, the trap is established not only in the user program, but in all system library routines included in the user field length as well. Since this can result in many unnecessary suspensions of execution, you should not use the * form when setting an RJ trap.

The RJ trap can be useful for checking input to and output from subroutines, as illustrated by the debug session shown in figure 3-8. This session was produced by executing the program shown in figure 3-4 in debug mode. Two SET,TRAP commands are issued: one for the main program PROG1 and one for subroutine SETB. The first trap suspends execution at each call to SETB and on initial entry into the main program. The second trap suspends execution at the RETURN statements in SETB. The PRINT command is entered after each trap occurrence to display the values passed to SETB and to display the values stored into the array B by SETB.

REMOVING TRAPS

A user-defined trap can be removed at any time during a debug session with the CLEAR,TRAP command. This command has the following forms:

CLEAR,TRAP
Remove all user-defined traps.

CLEAR,TRAP,type,scope
Remove the traps of the specified type established within the specified scope.

```
CYBER INTERACTIVE DEBUG
?set,trap,rj,p.main <-------------------- Set RJ trap in MAIN.

INTERPRET MODE TURNED ON
?set,trap,rj,p.setb <-------------------- Set RJ trap in SETB.

?go <------------------------------------ Initiate execution.

*T #1, RJ IN L.0 <----------------------- Hidden return jump on initial entry into main program.
?go

*T #1, RJ IN L.5 <----------------------- Trap occurs at line 5 of main program.
?print*,n,k

 5 1
?go

*T #2, RJ IN P.SETB_L.10 <--------------- RJ trap at line 10 of SETB.
?print*,b

 1. 1. 1. 1. 1.
?go

*T #1, RJ IN P.MAIN_L.7 <---------------- RJ trap at line 7 of MAIN.
?print*,k

 2
?go

*T #2, RJ IN P.SETB_L.6 <---------------- RJ trap at line 6 of SETB.
?print*,b

 -1. -1. -1. -1. -1.
?go

*T #17, END IN P.MAIN_L.8
?
     STOP
       .568 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Figure 3-8. Debug Session Illustrating RJ Trap

CLEAR,TRAP,type
Remove all traps of the specified type.

CLEAR,TRAP,#n_1,#n_2,...
Remove the traps identified by the specified numbers.

The type parameter can be any of the types listed in table 3-1 except for the default INTERRUPT, END, and ABORT traps, which cannot be removed. The scope parameter is the same as for the SET,TRAP command (tables 2-1 and 2-2). An asterisk (*) can be specified for type or scope. If * is specified for type, all trap types are cleared; if * is specified for scope, unlimited scope is indicated.

The CLEAR,TRAP command can be used to remove traps that are no longer needed in a debug session. The command is also useful when editing command sequences, as discussed in section 5. Following are some examples of the CLEAR,TRAP command:

CLEAR,TRAP,RJ,P.GAUSS
Clear the RJ trap from program unit GAUSS.

CLEAR,TRAP,STORE,P.SUBX A
Clear the STORE trap at the variable A.

CLEAR,TRAP,*,L.10 ... L.100
Clear all traps defined for source lines 10 through 100 of the home program.

CLEAR,TRAP,JUMP
Clear all JUMP traps.

CLEAR,TRAP,#2,#4,#5
Clear the traps identified by trap numbers 2, 4, and 5.

A session log using the CLEAR,TRAP command is illustrated in figure 3-9. The session in figure 3-8 is duplicated except that the CLEAR,TRAP command is issued after the third pass through the loop, allowing the program to run to completion without interruption. Note that the default END trap is not removed by the CLEAR,TRAP command.

INTERPRET MODE

The RJ, JUMP, FETCH, and STORE traps require a mode of execution called interpret mode. In interpret mode, each machine instruction is simulated by an interpreter routine. Interpret mode is automatically activated when any of the preceding traps are established. Interpret mode remains on until all of these traps are cleared by a CLEAR,TRAP command or until explicitly turned off. CID indicates interpret mode by issuing the message:

INTERPRET MODE TURNED ON

Execution in interpret mode requires up to 50 times as much time as normal execution. When using traps that require interpret mode, you can reduce the amount of execution time required by turning interpret mode off when executing portions of the program not currently being debugged. Interpret mode is turned off by the command:

SET,INTERPRET,OFF.

Traps requiring interpret mode become inoperative if interpret mode is turned off. They can be reactivated by the command:

SET,INTERPRET,ON

```
CYBER INTERACTIVE DEBUG
?set,trap,store,c.bcom          <-- Set STORE trap for common block BCOM.

INTERPRET MODE TURNED ON
?go

*T #1, STORE INTO B IN P.SETB_L.8
?go                             <-- STORE trap occurs on first three passes through loop.
                                    Execution is resumed after each suspension.
*T #1, STORE INTO B+1 IN L.8
?go

*T #1, STORE INTO B+2 IN L.8
?clear,trap,store               <-- Remove trap after third pass through loop.

INTERPRET MODE TURNED OFF
?go                             <-- Resume execution.

*T #17, END IN P.MAIN_L.8
?
     STOP
       .345 CP SECONDS EXECUTION TIME

?quit


  DEBUG TERMINATED
```

Figure 3-9. Debug Session Illustrating CLEAR,TRAP Command

The use of the SET,INTERPRET command is illustrated by the debug session shown in figure 3-10. The program shown in figure 3-4 is executed in debug mode to produce this session. In this example, a STORE trap, which activates interpret mode, is established for the variable K in the main program. Interpret mode is then turned off while subroutine SETB is executing. To accomplish this, breakpoints are set at the beginning and at the end of the subroutine. When execution is suspended at the first breakpoint, interpret mode is turned off; when execution is suspended at the second breakpoint, interpret mode is turned back on, reactivating the STORE trap.

This method of turning off interpret mode is rather cumbersome since it is necessary to enter the SET,INTERPRET commands on each pass through the subroutine. This could be accomplished more efficiently by including the SET,INTERPRET commands in a command sequence so that they could be executed automatically. Command sequences are discussed in section 5.

SUMMARY OF TRAP AND BREAKPOINT CHARACTERISTICS

The following is a summary of trap and breakpoint information presented in this section:

- You can set or clear traps or breakpoints any time CID has control and has prompted for user input.
- Only one breakpoint can be established at a single statement; however, a single breakpoint and multiple traps can be set to occur at a single statement.
- Traps and breakpoints exist for the duration of the debug session, unless removed by the CLEAR command or inhibited by the SET,INTERPRET,OFF command before the session is terminated.
- The frequency parameters of the SET,BREAKPOINT command can be used to avoid suspending execution on each pass through a loop.
- CID provides END, ABORT, and INTERRUPT traps by default so that CID receives control on any program termination, even if you have not explicitly established any traps or breakpoints.
- Breakpoints suspend execution before the statement at the breakpoint location is executed. The point in the execution of a statement at which a trap suspends execution depends on the trap type. The statement at the trap or breakpoint location is executed in a normal manner.
- The RJ, JUMP, FETCH, and STORE traps activate INTERPRET mode, which increases execution time by as much as 50 times. To reduce execution time, specify the SET,INTERPRET,OFF command when executing portions of the program already debugged, or substitute breakpoints for these traps.
- When setting a JUMP or RJ trap, always specify a range for the scope parameter to avoid suspending execution on jump or return jump instructions occurring outside your program.

```
CYBER INTERACTIVE DEBUG
?set,trap,store,k          <-- Set STORE trap for variable K in main program.

INTERPRET MODE TURNED ON
?set,breakpoint,p.setb_1.3 <-- Set breakpoint at first executable statement of SETB

?set,breakpoint,p.setb_1.6 }
                           } <-- Set breakpoint at each RETURN statement.
?set,breakpoint,p.setb_1.9 }

?go

*T #1, STORE INTO K IN L.4
?go

*B #1, AT P.SETB_L.3       <-- Breakpoint detected at line 3 of SETB.
?set,interpret,off         <-- Turn off interpret mode.

?go

*B #3, AT P.SETB_L.9       <-- Breakpoint detected at line 9.
?set,interpret,on          <-- Turn on interpret mode.

?go

*T #1, STORE INTO K IN P.MAIN_L.6
?print*,k

 2
?quit

  DEBUG TERMINATED
```

Figure 3-10. Debug Session Illustrating SET,INTERPRET Command

DISPLAYING PROGRAM VARIABLES

When execution of your object program is suspended and CID has prompted for user input, you can enter commands to display the contents of program variables as they exist at the time of suspension. This discussion includes those forms of the display commands that are most useful to the FORTRAN programmer.

CID provides three commands for displaying the values of program variables: the LIST,VALUES command; the PRINT command; and the DISPLAY command. These commands are summarized in table 3-3.

LIST,VALUES COMMAND

The LIST,VALUES command alphabetically lists all variables defined in the source program and the current value of each. This command automatically formats the variables according to the variable type as declared in the source program. The command has the following forms:

LIST,VALUES
: List all variables and values defined in all program units, including arrays in their entirety.

LIST,VALUES,P.name$_1$,P.name$_2$,...
: List all variables and values defined in the specified program units.

The LIST,VALUES command can generate a large amount of output. In these cases, you can send the output to an auxiliary file (section 4) or use an alternate display command. It is generally more useful to PRINT or DISPLAY the values of specific variables.

An example of LIST,VALUES output is illustrated in figure 3-11. The program in figure 3-1 was executed in debug mode to produce this output. Execution is suspended at the CALL statement in the main program and at the RETURN statement in subroutine AREA, and a LIST,VALUES commands are issued. The symbol -I displayed as the value of the variable A indicates an indefinite value. (The value assigned to an initialized variable is determined by the installation.) This should be investigated, as it could be due to an initialization error. However, in this example, the indefinite occurs because the statement that stores a value into A has not yet been executed. The session is terminated after one pass through AREA.

TABLE 3-3. DISPLAY COMMANDS

Command	Description	Formatting	Scope
LIST,VALUES	Lists alphabetically all variable names and values within specified scope.	Automatic according to variable type.	Specified program unit; entire program if none specified.
PRINT	Displays contents of specified variables.	Automatic according to variable type.	Home program only.
DISPLAY	Displays contents of specified variable.	User-specified; default is variable type.	Default is home program; variables can be qualified for other than home program.

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.5

?set,breakpoint,p.area_l.8

?go

*B #1, AT L.5 <------------------------------ Execution suspended at line 5 of main program.
?list,values,p.rdtr <------------------------ Display variables in main program.

P.RDTR
A = -I,    X1 = 0.0,    X2 = .5,    X3 = -1.0,    Y1 = 1.0,    Y2 = 2.0
Y3 = 1.2 <----------------------------------- A value has not yet been stored into A.
?go

*B #2, AT P.AREA_L.8 <------------------------ Execution suspended at line 8 of AREA.
?list,values,p.area <------------------------ Display variables in AREA.

P.AREA
A = .54999999999999,    S1 = 1.1180339887499,    S2 = 1.01980390271 86
S3 = 1.7,    T = 1.9189189457342,    X1 = 0.0,    X2 = .5,    X3 = -1.0
Y1 = 1.0,    Y2 = 2.0,    Y3 = 1.2
?quit <-------------------------------------- Terminate session.

  DEBUG TERMINATED
```

Figure 3-11. Debug Session Illustrating LIST,VALUES Command

PRINT COMMAND

The PRINT command, introduced in section 2, is the most useful of the display commands for the FORTRAN programmer. This command is identical in format and function to the FORTRAN Extended list directed PRINT statement. The format is:

PRINT*,list

List elements must be separated by commas and can consist of any of the following:

Simple or subscripted variables

Array names

Constants

FORTRAN expressions not involving exponentiation or functions

Implied DO loops enclosed in parentheses

Qualified address forms cannot be used with the PRINT command. Except for variables declared in common, the PRINT command can only display variables local to the home program. To display variables belonging to another program unit you must designate a new home program with the SET,HOME command.

To print the contents of an array you can use the FORTRAN implied DO statement or you can simply specify the array name. For example, if the statement DIMENSION A(10) appears in the source program then the CID commands:

```
PRINT*,A
```

and

```
PRINT*,(A(I),I=1,10)
```

are equivalent. It should be noted, however, that in the case of multidimensioned arrays, specification of the array name causes the elements to be displayed in column order (the order in which they are stored), while the implied DO from can be used to specify a row-order display.

If the implied DO form is used, CID issues a warning message if the index exceeds the dimensioned boundaries of the array. The variable used as the index in the implied DO does not alter a variable of the same name used in the FORTRAN program.

The PRINT command automatically formats each variable according to its type as declared in the source program. To display variables in a format other than that declared in the source program you must use the DISPLAY command.

A session log illustrating the PRINT command is shown in figure 3-12. Execution of subroutine SETB (figure 3-4) is suspended at lines 6 and 9 by a breakpoint. The PRINT command displays the contents of the array B, the contents of the variables N and K, and the value of 3*K.

DISPLAY COMMAND

The DISPLAY command displays the contents of specified locations within a program. In most cases, you will be using the PRINT command since it provides for automatic formatting of variables and is more familiar to FORTRAN programmers. The DISPLAY command, however, offers the following advantages:

- DISPLAY can display values belonging to any program unit, while PRINT can only list values local to the home program.
- DISPLAY allows you to specify the format of each variable, while PRINT performs automatic formatting. In most cases, automatic formatting is more convenient. However, the PRINT command cannot display character data. You can do this with DISPLAY by specifying character format. In addition, DISPLAY allows you to examine data in its internal representation by specifying octal format.
- DISPLAY is the only command that can display the contents of debug variables. (See section 4.)

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,p.setb_1.6

?set,breakpoint,p.setb_1.9

?go

*B #2, AT P.SETB_L.9
?print*,"values are ",(b(i),i=1,n),n,k,3*k <-- Print specified values while execution is suspended at line 9 of SETB.

 VALUES ARE 1. 1. 1. 1. 1. 5 1 3
?go

*B #1, AT P.SETB_L.6
?print*,"b = ",b," k = ",k <-- Print specified values while execution is suspended at line 6 of SETB.

 B = -1. -1. -1. -1. -1.  K = 2
?quit

  DEBUG TERMINATED
```

Figure 3-12. Debug Session Illustrating PRINT Command

DISPLAY can be used to display the contents of any program location, although you will usually use it to display the contents of program variables. The format of the DISPLAY command is:

```
DISPLAY,location,format,count
```

location — Can be any of the formats listed in tables 2-1 and 2-2.

format — Optional format designator; valid values are as follows:

F	Floating point
D	Double precision floating point
I	Integer
C	Character
O	Octal
A	Symbolic address (L.n or S.n)

Default is variable type as declared in the program.

count — Optional integer specifying the number of consecutive locations to display; default is 1.

A major disadvantage of the DISPLAY command is that a list of variables cannot be specified. For example, to display the contents of the variables A, B, and C requires the three DISPLAY commands:

```
DISPLAY,A
DISPLAY,B
DISPLAY,C
```

However, this can be accomplished with the single PRINT command:

```
PRINT*,A,B,C
```

Unlike the PRINT command, the DISPLAY command displays only the first word when an array name is specified. To display the contents of an array you must either use ellipsis notation to specify a range or specify the count parameter. For example, if the statement DIMENSION A(10) appears in the source program, then the commands:

```
DISPLAY,A...A+9
DISPLAY,A,,10
```

are equivalent; both display the values of all 10 words of A.

Since the DISPLAY command automatically formats variables, it is necessary to specify the format parameter only when you want to display the variable in a format other than that declared in the FORTRAN program.

Figure 3-13 illustrates a program and a debug session in which the DISPLAY command is used to display character data. The program reads records containing a person's name and the state in which he lives. After each record is read, the state name is tested; if it is California, the person's name is printed. The program then reads the next record. When a ZZZZ is detected as the first word of a record, the program terminates.

In the debug session, a breakpoint is initially set at line 7 to suspend execution after each record is read. After each suspension the DISPLAY command is used to display the values of the input variables in character format. Note that the PRINT command, issued after the first read, converts the values to type integer.

Figure 3-14 illustrates a program and debug session in which the DISPLAY command is used to examine variables in their internal representation. The program reads groups of four integers from the terminal and packs each group into a single word using the SHIFT function and OR operator. Each packed integer occupies a 15-bit field. The program is executed under CID control and allowed to terminate after three sets of integers have been input. When the END trap gives control to CID, the DISPLAY command is used to display the first three words of the array IPACK in octal format showing the four 15-bit fields of each word. The PRINT command cannot be used since each word of IPACK would be automatically formatted as a decimal integer.

ALTERING PROGRAM VALUES

CID provides several commands to alter the contents of program locations. Although these commands can be used to change the contents of any location within the program field length, as a FORTRAN programmer you will usually be concerned only with changing the contents of variables.

CID provides three commands for altering program values:

- Assignment command. This command is identical to the FORTRAN assignment statement. It allows you to evaluate expressions and to insert values into variables in the home program.
- ENTER command. The function of this command is similar to the assignment command; however, it is less powerful than the assignment command and is therefore seldom used by FORTRAN programmers.
- MOVE command. This command can be used to move data from one program unit to another.

Since the assignment command largely precludes the need for the ENTER command, ENTER is not discussed here. Refer to the CYBER Interactive CID reference manual for details on the ENTER command.

ASSIGNMENT COMMAND

The assignment command is identical in form and function to the FORTRAN assignment statement. This command allows you to make corrections to your program as execution proceeds, eliminating the need for recompiling each time an error is discovered. The assignment command has the form:

```
var=expression
```

where var is a simple or subscripted variable and expression is any valid FORTRAN arithmetic expression not involving functions or exponentiation. The assignment command functions exactly as in FORTRAN: The expression is evaluated and its value is assigned to the variable on the left of the equal sign; the previous contents of the receiving variable are destroyed. You can enter an assignment command whenever CID has prompted for user input. For example, if program execution is suspended and you have detected a variable that has an incorrect or illegal value, you can use the

Program CHAR:

```
1             PROGRAM CHAR       73/74   TS ID


                                 PROGRAM CHAR(INFILE,TAPE2=INFILE,OUTPUT)
                                 REWIND 2
                                 WRITE 200
                          200    FORMAT (" OUTPUT",////)
          5               10     READ (2,300) NAME,ISTATE
                          300    FORMAT (A10,A2)
                                 IF(NAME.EQ.4HZZZZ)STOP
                                 IF(ISTATE.NE.2HCA) GO TO 10
                                 WRITE 400,NAME
         10               400    FORMAT (" ",A10)
                                 GO TO 10
                                 END
```

Input Data:

```
FIDDLE     CA
BIDDLE     NY
GRIDDLE    CALIF
DIDDLE     WASH
ZZZZ
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,1.7

?go

OUTPUT



*B #1, AT L.7
?print*,name,istate

 1106375458920108 61 54525724059949933
?display,name,c

NAME = FIDDLE
?display,istate,c

ISTATE = CA
?go

FIDDLE
*B #1, AT L.7
?display,name,c

NAME = BIDDLE
?display,istate,c

ISTATE = NY
?quit

  DEBUG TERMINATED
```

PRINT command formats according to variable type.

Display NAME and ISTATE in character format.

Read another input record, display input values, and terminate session.

Figure 3-13. Program CHAR, Input File, and Debug Session Illustrating DISPLAY Command

```
1             PROGRAM PAK        74/74   TS ID

                          PROGRAM PAK(INPUT,OUTPUT)
                          DIMENSION IPACK(10),I(4)
                          N=1
                     4    WRITE 100
          5          100  FORMAT(" INTEGERS? ")
                          READ 300, (I(J),J=1,4)
                     300  FORMAT(4I3)
                          IF (I(1).EQ.999) STOP
                          IPACK(N)=SHIFT(I(4),45).OR.SHIFT(I(3),30).OR.
         10              X     SHIFT(I(2),15).OR.I(1)
                          IF(N.GE.10) STOP
                          N=N+1
                          GO TO 4
                          END

CYBER INTERACTIVE DEBUG
?go

INTEGERS?  4   8 12   1
INTEGERS? 25   6 14  31
INTEGERS? 18 14   7  10
INTEGERS?999

*T #17, END IN L.8
?
     STOP
       .102 CP SECONDS EXECUTION TIME
display,ipack,o,3

IPACK = 00001 00014 00010 00004    00037 00016 00006 00031
    " +2 = 00012 00007 00016 00022
?quit

  DEBUG TERMINATED
```

Input three sets of integers and an end-of-input indicator.

END trap occurs on program termination.

Display three words of array IPACK in octal format, showing packed integers.

Figure 3-14. Program PAK and Debug Session Illustrating DISPLAY Command

assignment command to assign a new value to the variable. When you resume execution of the program, the new value is used in subsequent computations involving the altered variable.

Expressions used in assignment commands can be any valid FORTRAN expression with the exception of function references and exponentiation. Any valid FORTRAN constant can appear in an expression. The assignment command performs all conversions according to the rules of FORTRAN. An assignment command cannot span more than one line.

The variables used in an assignment command must all be defined in the home program. To reference variables in another program unit you must specify the SET,HOME command to designate that program unit as the home program. Just as with FORTRAN, variables are local to the program unit in which they are defined and cannot be mixed in an assignment command with variables local to another program unit.

Changes made through the assignment command do not exist beyond the end of the debug session. When a program is reexecuted, either in debug mode or in normal mode, all program variables have the values defined in the original compiled version.

Following are some examples of assignment commands:

A=B
: Replace the current contents of A by the current contents of B.

M=N+I-1
: Evaluate the expression using the current contents of N and I and assign the value to M.

ARR(I)=X(I)*X(I)+4./3.*(Y+Z)*2.
: Evaluate the expression using the current values of X, I, Y, and Z and assign the value to the Ith word of ARR.

The sample program shown in figure 3-15 is executed under CID control to illustrate the assignment command. The program calculates the mean of 10 numbers. The program contains a bug: the statement AV=SUM*10.0 should be AV=SUM/10.0.

To enable the program to execute correctly, a breakpoint is set at the WRITE statement. When execution is suspended at this location, the program has already calculated an incorrect value for AV. The assignment command is then used to calculate the correct value of AV. The new value is used in the subsequent WRITE

Program AVG:

```
1             PROGRAM AVG        74/74   TS ID


                          PROGRAM AVG(OUTPUT)
                          DIMENSION X(10)
                          DATA X/1.0,15.3,2.4,12.7,6.0,
                         *       5.5,10.1,9.4,4.8,2.0/
         5                SUM=0.0
                          DO 12 I=1,10
                          SUM=SUM+X(I)
                    12    CONTINUE
                          AV=SUM*10.0
        10                WRITE 100, (X(I),I=1,10),AV
                    100   FORMAT(" NUMBERS: ",10F5.2,/" MEAN: ",F5.2)
                          STOP
                          END
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.10 <-------- Set breakpoint to suspend execution at WRITE statement.

?go

*B #1, AT L.10
?print*,av <-------- Print value of AV calculated in the program.

 691.99999999999
?av=sum/10.0 <-------- Calculate correct value for AV.

?print*,av

 6.9199999999999
?go

NUMBERS:  1.0015.30 2.4012.70 6.00 5.5010.10 9.40 4.80 2.00
MEAN:  6.92 <-------- Program writes correct value.
*T #17, END IN L.12
?
     STOP
       .251 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Figure 3-15. Program AVG and Debug Sessions Illustrating Assignment Command

statement when execution is resumed. The erroneous statement will be replaced by the user in the final version of the source program.

Some additional examples of assignment commands are illustrated in the examples at the end of this section.

CONDITIONAL EXECUTION OF ASSIGNMENT COMMANDS

The IF command allows for conditional execution of a CID command. It is identical in form and function to the standard form of the FORTRAN logical IF statement. The IF command has the form:

IF(expr) command

expr	Any valid FORTRAN conditional expression not involving functions or exponentiation.
command	Any valid CID command.

If the conditional expression is true, the CID command is executed.

The IF command is extremely powerful when used in command sequences. (See section 5.) However, the IF command can also be useful in interactive mode, as illustrated by the following example. Consider the program segment:

```
READ 100,A,B,C,Y
X=Y/(A+B-C*2.0)
    .
    .
    .
```

If the denominator has a value of zero, an illegal operand is generated. Abnormal termination can be prevented during the debug session by testing the denominator and assigning X an arbitrary value if the denominator is zero. This can be done by setting a breakpoint at the statement immediately following the computation of X and, when the breakpoint is reached during execution, entering the command:

```
IF(A+B-C*2.0 .EQ.0.) X=1.0
```

The infinite value generated for X is replaced by 1.0, and the program will not abort when X is used as an operand in subsequent computations.

MOVE COMMAND

The MOVE command moves data from one location to another. In most cases, you will use the assignment command to alter the contents of program variables (insert or move data). However, the MOVE command facilitates the moving of blocks of data. In addition, MOVE is the only command that can move data between program units. The forms of the MOVE command are:

MOVE,$range_1$,$range_2$
: Move a block of data from $range_1$ to $range_2$. $Range_n$ must be one of the range specifications listed in table 2-2, other than a program unit name. If $range_1$ is greater than $range_2$, only enough data is moved to fill $range_2$. If $range_2$ is greater than $range_1$, the block of data in $range_1$ is moved repeatedly until $range_2$ is filled.

MOVE,$address_1$,$address_2$,n
: Move n data items from consecutive locations starting at $address_1$ to consecutive locations starting at $address_2$. If n is omitted, the default is 1.

The range parameters are usually specified using ellipsis notation. For example,

```
MOVE,A...A+9,B...B+9
```

moves ten consecutive words from locations starting at A to locations starting at B.

The MOVE command is useful for moving data between arrays and between common blocks. Arrays can be in different program units.

The following examples assume a program containing common blocks ACOM and BCOM, and subroutines named SUBA and SUBC; SUBA contains the statements DIMENSION A(100) and DIMENSION B(100); SUBC contains the statement DIMENSION C(100).

MOVE,A,B
: Move the contents of the first word of A to the first word of B.

MOVE,A,B,100
: Move the contents of A to corresponding locations in B.

MOVE,A...A+99,B...B+99
: Move the contents of A to corresponding locations in B.

MOVE,A...A+1,B...B+6
: The results of this move are as follows:

Contents of	Moved to
A(1)	B(1)
A(2)	B(2)
A(1)	B(3)
A(2)	B(4)
A(1)	B(5)
A(2)	B(6)

MOVE,P.SUBA_A,P.SUBC_C,20
: Move the contents of the first 20 locations of A to the corresponding locations of C.

MOVE,A+2...A+5,B...B+3
: Move the contents of A(3) through A(6) to B(1) through B(4).

MOVE,C.ACOM,C.BCOM,10
: Move the contents of the first 10 locations of ACOM to the first 10 locations of BCOM.

DEBUGGING EXAMPLES

The following paragraphs present some examples of interactive debugging using the commands discussed in this section.

SAMPLE PROGRAM CORR

The program entitled CORR reads pairs of numbers and calculates the correlation coefficient of the numbers. The source listing is shown in figure 3-16.

The correlation coefficient is a means of measuring the degree of statistical correlation between two sets of numbers. The formula for the correlation coefficient is:

$$r = \frac{n\Sigma xy - \Sigma x \Sigma y}{\sqrt{n\Sigma x^2 - (\Sigma x)^2}\ \sqrt{n\Sigma y^2 - (\Sigma y)^2}}$$

r	correlation coefficient
n	number of pairs to be correlated
x,y	values to be correlated

The correlation coefficient can have any value between -1 and 1. A coefficient with magnitude close to 1 indicates close correlation.

The program in figure 3-16 contains a number of bugs. It is assumed that the programmer has performed a visual scan of the program and is ready to attempt an execution.

To execute the program, some test data is required. If possible, test cases should be included for which results are known. In the example, the first test case consists of pairs of equal numbers; if the program is correct it should calculate a correlation coefficient of 1.0.

```
1            PROGRAM CORR       74/74   TS ID

                      PROGRAM CORR(OUTPUT,CORFIL,TAPE2=CORFIL)
                      DIMENSION X(5),Y(5)
                C
                C...INITIALIZATION
 5              C
                      N=1
                      SUMX=0.0
                      SUMY=0.0
                      SUMXSQ=0.0
10                    SUMXY=0.0
                C
                C...READ NUMBERS TO BE CORRELATED
                C
                      REWIND 2
15              10    READ(2,*) X(N),Y(N)
                      IF(EOF(2).NE.0) GO TO 20
                      N=N+1
                      GO TO 10
                C
20              C...CALCULATE CORRELATION COEFFICIENT
                C
                20    IF(N.EQ.0) GO TO 50
                      DO 30 I=1,N
                      SUMX=SUMX+X(I)
25                    SUMY=SUMY+Y(I)
                      SUMXSQ=SUMXSQ+X(I)**2
                      SUMYSQ=SUMYSQ+Y(I)**2
                      SUMXY=X(I)+Y(I)
                30    CONTINUE
30                    NUM=(N*SUMXY-SUMX*SUMY)**2
                      DENOM=(N*SUMXSQ-SUMX**2) * (N*SUMYSQ-SUMY**2)
                      RSQ=NUM/DENOM
                      R=SQRT(RSQ)
                      WRITE 800,R
35              800   FORMAT(" CORRELATION COEFFICIENT IS ",F6.2)
                      STOP
                50    WRITE 810
                810   FORMAT(" EMPTY INPUT FILE")
                      STOP
40                    END
```

Figure 3-16. Program CORR Before Debugging

Since it is anticipated that the program will contain errors, CID is used for the first attempt at execution. The strategy is to allow the program to execute as far as possible, and to then use information obtained during the initial session to determine where to set traps and breakpoints for subsequent sessions.

The initial attempt to execute CORR is shown in figure 3-17. For the first session, no traps or breakpoints are established; GO is issued to initiate program execution and the program is simply allowed to execute until termination, providing interactive control via an ABORT trap. The trap message indicates that an execution error has occurred. Error 04 is caused by a computation involving an indefinite operand. Line 27 of the source program contains the statement SUMX =SUMY + Y(I).

Since the ABORT trap has given control to CID, commands can be entered to try and determine the cause of the error. The PRINT command displays: the contents of the arrays X and Y, into which the input values are stored; and the variable N, containing the number of cards read. The value of the index I is displayed since execution terminated within a DO loop. Note that only the first five pairs of values are displayed (the X and Y arrays are dimensioned 5). This indicates a possible error because the test case contains 10 cards. Also, the values of the card counter and the DO loop index are both equal to 11. Suspicions of an indexing error are verified by the next PRINT command, which uses an implied DO to print the contents of X. The program contains no check on the number of records read, which allowed the array bounds to be exceeded. When the source program is corrected, such a check will be included; but so that debugging can continue without recompiling, the extra data cards are simply removed from the input deck and the program is rerun.

The second debug session is shown in figure 3-18. Abnormal termination again occurs at line 27. Since execution terminated within the loop, the values of I and N are again printed along with some intermediate values. Although the data file contains only 5 records, the counter N has a value of 6. This has caused another indexing error. Referring to the source listing, it is seen that N is initialized to 1 and is incremented after each card is read; N will always contain a value one greater than the actual number of records read.

```
Input Data:

  1.0 1.0
  10.0 10.0
  7.6 7.6
  2.9 2.9
  5.1 5.1
  3 3
  100.5 100.5
  7.0 7.0


Session Log:

  CYBER INTERACTIVE DEBUG
  ?go

  *T #18, ABORT CPU ERROR EXIT 04 IN L.27 <---- ABORT trap at line 27.
  ?print*,x,y,n,i

   1. 5.1 7.6 10. 2.4 3. 7. 100.5 10. 2.4 9 1
  ?print*,(x(i),y(i),i=1,n) <---- This form of the PRINT command indicates a subscript error.

  *WARN - SUBSCRIPT OUT OF RANGE
  OK ?quit

    DEBUG TERMINATED
```

Figure 3-17. Input Data for First Test Case and Debug Session

```
CYBER INTERACTIVE DEBUG
?go

*T #18, ABORT CPU ERROR EXIT 04 IN L.27 <---- ABORT trap at line 27.
?print*,n,sumx,sumy,sumxsq,sumysq,sumxy,i <---- Display intermediate values.
                                          <---- N exceeds array boundary.
 6 1. 1. 1. -I 0. 1
?quit                                     <---- SUMYSQ contains indefinite value.

  DEBUG TERMINATED
```

Figure 3-18. Second Debug Session

The counter can be corrected by initializing it to 0 instead of 1. The CID output also shows that SUMYSQ contains an indefinite value. This is caused by failure to initialize SUMYSQ to 0. Without changing the source code and recompiling, debugging can continue by conducting another debug session and using assignment commands to insert the correct values for N and SUMYSQ.

The third debug session is shown in figure 3-19. When the breakpoint at line 20 occurs, N is set to 0 and SUMYSQ is set to 0. The GO command resumes execution; this time the program runs to completion. The value displayed for R, however, is clearly incorrect since the correct value is known to be 1.0. The next PRINT command shows that all the data values are being read correctly, and it is known from the second session that all the intermediate sums are correctly initialized. Another session will be conducted this time with execution suspended at various points within the computation portion of the program so that the progress of the calculations can be examined.

The fourth session is shown in figure 3-20. The correct initial values for N and SUMXY are inserted as in the previous session.

A JUMP trap is set at line 29, to suspend execution on each pass through the loop, and a breakpoint is set at line 33, to suspend execution immediately prior to execution of the statement R=SQRT(RSQ).

Suspension at these statements allows all intermediate values to be examined and any last minute changes made before calculation of the final result. After the trap and breakpoints are set, GO is entered. CID then issues an ABORT message indicating that the allowable time limit has been exceeded. The JUMP trap activated interpret mode, which caused program execution to exceed the system default time limit. The current session is terminated and another session is initiated. This time a breakpoint, instead of a JUMP trap, is set at line 29. The breakpoint suspends execution before the instructions

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.22 <-- Set breakpoint at line 22.

?go

*B #1, AT L.22
?print*,n,sumysq

 6 -I
?n=n-1
                      <-- Calculate correct values for N and SUMYSQ.
?sumysq=0.0

?go

CORRELATION COEFFICIENT IS    2.42 <-- Final result is incorrect.
*T #17, END IN L.36
?
     STOP
       .296 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Figure 3-19. Third Debug Session

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.22

?set,trap,jump,l.29

INTERPRET MODE TURNED ON
?set,breakpoint,l.33

?go

*T #18, ABORT CP TIME LIMIT IN L.15 <-- ABORT trap at line 15; time limit exceeded.
?quit

  DEBUG TERMINATED
..rewind,lgo <-- Rewind binary file.

..lgo <-- Initiate new Debug session.

CYBER INTERACTIVE DEBUG
?set,breakpoint,l.22
                      <-- Set breakpoints at lines 22, 29, and 33.
?set,breakpoint,l.29
?set,breakpoint,l.33

?go

*B #1, AT L.22
?n=n-1
                      <-- Calculate correct values for N and SUMYSQ.
?sumysq=0.0
```

Figure 3-20. Fourth Debug Session (Sheet 1 of 2)

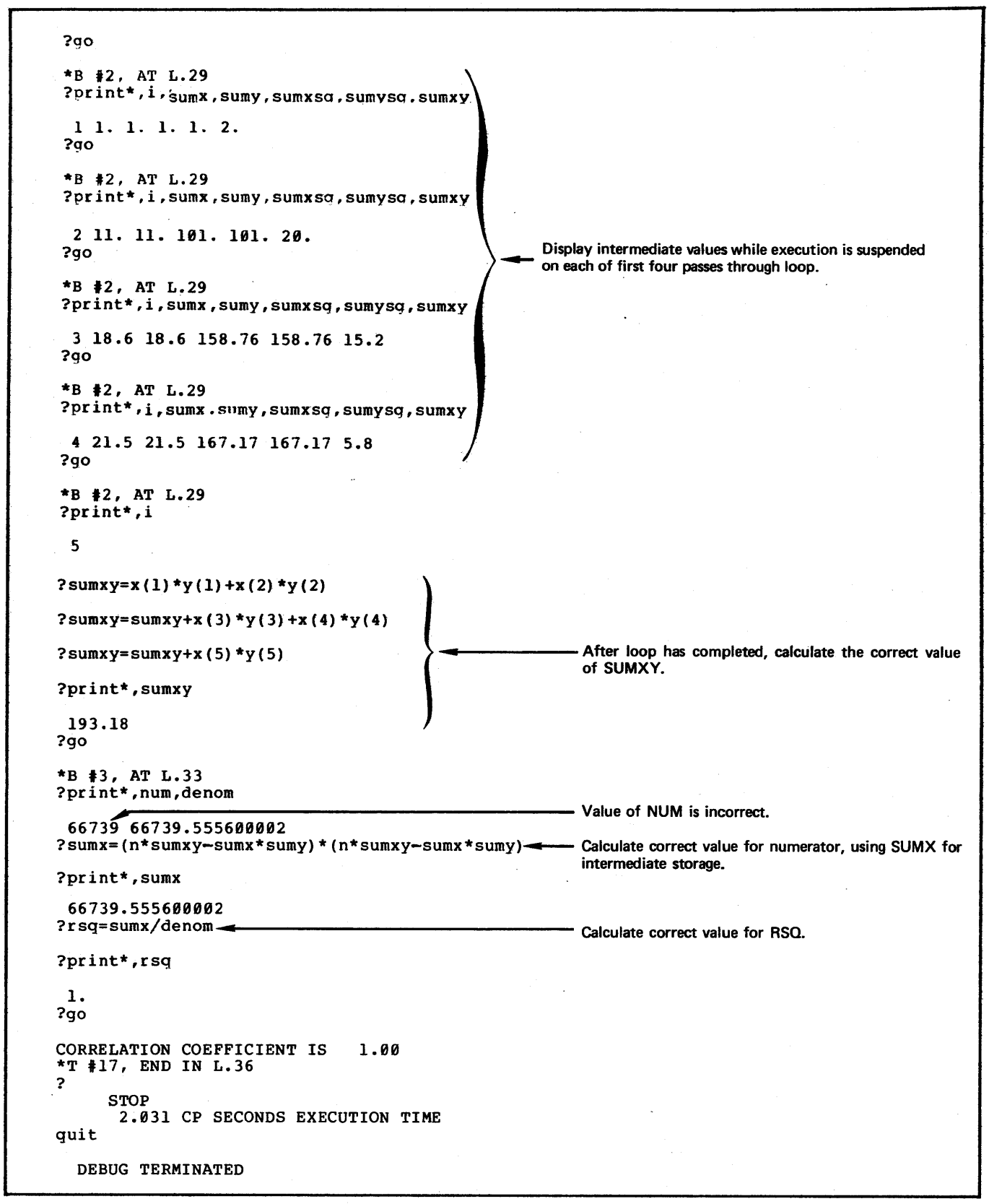

Figure 3-20. Fourth Debug Session (Sheet 2 of 2)

generated by line 29 are executed, but since this is a CONTINUE statement, the breakpoint has the same effect as the JUMP trap.

The session is then conducted as the preceding one, with correct initial values inserted for N and SUMXY before execution is initiated. The first suspension occurs at line 29, after the first pass through the loop. A printout of the intermediate sums at this point does not indicate that anything is amiss. Two more passes through the loop still do not indicate an error, at least not at first glance. On the fourth pass, however, the value of SUMXY does not appear to be consistent. SUMXY contains a sum of positive numbers, and yet its value on the fourth pass is less than its value on the third pass. The statement SUMXY = X(I) + Y(I) in the source program is incorrect; it should read SUMXY = SUMXY + X(I) * Y(I).

The debug session can be continued by using the assignment command to calculate and insert the correct value of SUMXY. First, execution is resumed to allow the loop to complete. After the last pass through the loop, the correct value is calculated with an appropriate assignment command. The next suspension occurs at line 33, where a breakpoint was set. The value of RSQ is printed and is clearly wrong; since the correct value of R is known to be 1.0, the square of R should also be 1.0. The next step is to examine the values used to calculate RSQ: NUM and DENOM. For RSQ to have a value of 1.0, NUM and DENOM must be equal. However, the PRINT command shows that NUM and DENOM are not equal. NUM is implicitly an integer and when the floating point value was stored into NUM, truncation occurred. When the source program is corrected, the name NUM will be replaced by a name that is type REAL. The session can be continued, however, by once again using an assignment command to calculate the correct value for the numerator. This requires a temporary location into which the value of the numerator can be stored. The variable SUMX can be used for this temporary location since it is not referenced after line 33. After the numerator is calculated and stored in SUMX, an assignment command is used to calculate RSQ. The PRINT command shows that RSQ now has the correct value. Execution is resumed at line 33, the location where it was suspended; line 33 calculates the final result. The program runs to completion and the session is terminated by QUIT. The program now appears to execute correctly, at least for this particular test case.

At this point, it is probably a good idea to incorporate all the accumulated changes into the source deck, recompile, and rerun the program to verify the corrections. However, the program cannot be considered completely debugged until it has been tested on additional sets of input data.

For the next test case data records are included in which all the X values are equal. The input file and session log are shown in figure 3-21. The program runs to completion but an error occurs in the SQRT routine and the indefinite character I is printed for the correlation coefficient. Using CID commands to display intermediate values, it is seen that a division by zero has occurred. CID has helped determine the location of the error, but in order to understand why the error occurred it is necessary to understand the mathematics of the program.

In the formula for the correlation coefficient it can be shown that the calculation $n \Sigma x^2-(\Sigma x)^2$ has a value of zero if all the x values are equal. Whenever a division occurs within a program, you should always be alert to the possibility of a zero denominator and include statements testing for that possibility.

Input Data:

```
3.0 1.0
3.0 5.1
3.0 7.6
3.0 10.0
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?go

ARGUMENT INFINITE
ERROR NUMBER   39   DETECTED BY SQRT
CORRELATION COEFFICIENT IS       I
*T #17, END IN L.37
?
     STOP
       .106 CP SECONDS EXECUTION TIME
print*,rsq,anum,denom
 R 3.30872245021211E-24 0.
?quit

  DEBUG TERMINATED
```

Error detected in SQRT routine.

RSQ is out of range.

Figure 3-21. Input Data for Second Test Case and Debug Session

To complete the debugging process two more test cases are run: one in which the data correlates closely (figure 3-22) and one in which the values are widely scattered (figure 3-23). The results of both tests appear to be correct. In a real situation correctness of the results should be verified whenever possible by comparing with known results or by performing hand calculations. The final version of CORR, with all corrections included, is shown in figure 3-24.

Input Data:

```
10.1 10.1
20.5 21.1
6.0 6.0
34.0 32.9
4.4 4.5
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?go

CORRELATION COEFFICIENT IS    1.00
*T #17, END IN L.39
?
      STOP
        .100 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Figure 3-22. Input Data for Third Test Case and Debug Session

Input Data:

```
10.0 -6.4
2.0 15.2
4.3 1.1
2.8 10.0
5.5 -3.7
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?go

CORRELATION COEFFICIENT IS     .22
*T #17, END IN L.39
?
      STOP
        .116 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Figure 3-23. Input Data for Fourth Test Case and Debug Session

SAMPLE PROGRAM NEWT

Program NEWT finds a zero root of a functon by Newton's method. Newton's method generates successive approximations to the equation f(x)=0 by applying the iteration:

$$x_{i+1}=x_i-f(x_i)/d(x_i)$$

where:

$f(x_i)$ is the current functional value.

$d(x_i)$ is the derivative of the current functional value.

i is the current approximation to the root.

i+1 is the new approximation to the root.

The program listing, shown in figure 3-25 contains line numbers generated by the NOS line editor.

To use Newton's method, you start with an initial approximation and apply the preceding scheme to calculate a new, better approximation. You then substitute the new approximation into the relation and calculate a still closer approximation. Each successive approximation is closer to the desired root. The process is continued until the desired degree of accuracy is achieved.

The program to implement Newton's method consists of a main routine, a subroutine to apply Newton's method, and two function subprograms: F, which defines the function to be solved, and D, which calculates the derivative of the function.

The main program passes an initial approximation of the solution to subroutine NEWT, along with the function names. NEWT initializes an error flag IER, and a variable ITS which contains the current number of iterations. The iterative scheme is applied in lines 400 through 480. If the initial approximation is itself a zero root, control returns to the main program. A zero derivative generates an error; therefore, a test is included for a zero value of the function D. Line 440 calculates a new approximation X. Line 410 tests the functional value FX of the current approximation; if FX has a value of zero, control returns to the main program. This type of test, as will be shown, causes problems when used with an iterative scheme.

Subroutine NEWT returns the value of the solution X, the number of iterations required ITS, and the error flag IER.

The function to be solved, defined in lines 550 through 590, is:

$$f(x)=3.0x-(x+1.0)/(x-1.0)$$

The derivative of the function, lines 610 through 670, is:

$$d(x)=3.0+2.0/(x-1.0)^2$$

The debug session for this program is shown in figure 3-26. Since the program involves several subprograms, some traps that should be helpful in the debugging process are set initially. The RJ trap in the main program and in subroutine NEWT will suspend execution at the subroutine and function calls and returns in those program units. The FETCH and STORE traps, established at location zero, will detect a subroutine call with too few arguments.

```
1             PROGRAM CORR       73/74   TS ID

                      PROGRAM CORR(OUTPUT,CORFIL,TAPE2=CORFIL)
                     †DIMENSION X(5),Y(5)
                C
                C...INITIALIZATION
 5              C
                     †N=0
                      SUMX=0.0
                      SUMY=0.0
                      SUMXSQ=0.0
10                    SUMYSQ=0.0
                     †SUMXY=0.0
                C
                C...READ NUMBERS TO BE CORRELATED
                C
15                    REWIND 2
                10   †READ(2,*) X(N+1),Y(N+1)
                      IF(EOF(2).NE.0) GO TO 20
                      N=N+1
                     †IF(N.GT.5) GO TO 40
20                    GO TO 10
                C
                C...CALCULATE CORRELATION COEFFICIENT
                C
                20    IF(N.EQ.0) GO TO 50
25                    DO 30 I=1,N
                      SUMX=SUMX+X(I)
                      SUMY=SUMY+Y(I)
                      SUMXSQ=SUMXSQ+X(I)**2
                      SUMYSQ=SUMYSQ+Y(I)**2
30                   †SUMXY=SUMXY+X(I)*Y(I)
                30    CONTINUE
                     †ANUM=(N*SUMXY-SUMX*SUMY)**2
                      DENOM=(N*SUMXSQ-SUMX**2) * (N*SUMYSQ-SUMY**2)
                     †IF(DENOM.EQ.0.0) GO TO 60
35                    RSQ=ANUM/DENOM
                      R=SQRT(RSQ)
                      WRITE 800,R
                800   FORMAT(" CORRELATION COEFFICIENT IS ",F6.2)
                      STOP
40              C
                C..†ERROR PROCESSING
                C
                40    WRITE 805
                805   FORMAT("TOO MUCH INPUT DATA. LIMIT IS 5 PAIRS"
45                    STOP
                50    WRITE 810
                810   FORMAT(" EMPTY INPUT FILE")
                      STOP
                60    WRITE 815
50              815   FORMAT(" BAD INPUT.  ALL X'S OR ALL Y'S MUST N
                      END
```

†Indicates changes.

Figure 3-24. Program CORR With Corrections

The first trap occurs on initial entry into the main program. The next one occurs at the call to NEWT. This is a good time to examine values input to the subroutine. If the program does not contain a large number of variables, LIST,VALUES is a good way to determine if variables have not been properly defined. The variable X0 has been initialized to zero; the variables IER and ITS have not been initialized, since their values are calculated in the subroutine. FF, which contains an indefinite value, is a misspelling of the function name F. Note that the display commands do not list subprogram names (the function name D does not appear in the list). The subsequent PRINT command also shows the undefined variables. At this point it might be evident that an argument was omitted from the CALL statement. This error is detected by STORE trap at line 370 which indicates that a value was stored into a formal parameter without a corresponding argument in the CALL statement. Storing a value into IER caused the trap to occur (IER is the sixth formal parameter in the subroutine statement; there are only five arguments in the CALL statement). The variable X0 is missing from the CALL argument list. An assignment command is used to assign the correct value to X0.

```
00100 PROGRAM MAIN(OUTPUT)
00110 EXTERNAL F,D
00120 X0=0.0
00130 CALL NEWT(FF,D,X,ITS,IER)
00140 IF(IER.NE.0) GO TO 900
00150 WRITE 100, ITS,X
00160 100 FORMAT(" CONVERGENCE IN ",I4," ITERATIONS.  X= ",E12.4)
00170 STOP
00180 900 STOP "ERROR IN SUBROUTINE NEWT"
00190 END
00200C
00210C   SUBROUTINE NEWT FINDS A ZERO ROOT OF AN EQUATION BY
00220C   NEWTONS METHOD
00230C
00240C   INPUT
00250C      F    NAME OF FUNCTION DEFINING EQUATION TO BE SOLVED
00260C      D    NAME OF FUNCTION DEFINING DERIVATIVE
00270C      X0   INITIAL APPROXIMATION TO ROOT
00280C
00290C   OUTPUT
00300C      X    SOLUTION TO F(X)=0
00310C      ITS  NUMBER OF ITERATIONS REQUIRED FOR SOLUTION
00320C      IER  ERROR FLAG
00330C           0  NO ERRORS
00340C           1  ERRORS
00350C
00360 SUBROUTINE NEWT(F,D,X0,X,ITS,IER)
00370 IER=0
00380 ITS=0
00390 X=X0
00400 10 FX=F(X)
00410 IF(FX.EQ.0.0) RETURN
00420 DX=D(X)
00430 IF(DX.EQ.0.0) GO TO 900
00440 X=X-FX/DX
00450 ITS=ITS+1
00460 GO TO 10
00470 900 WRITE 200, X0
00480 200 FORMAT(" DERIV 0 AT ",F6.2," SPECIFY DIFFERENT X0")
00490 IER=1
00500 RETURN
00510 END
00520C
00530C   F DEFINES A FUNCTION TO BE SOLVED BY NEWTONS METHOD
00540C
00550 FUNCTION F(X)
00560 IF(X.EQ.1.0) STOP "BAD ARG TO F"
00570 F=3.0*X-(X+1.0)/(X-1.0)
00580 RETURN
00590 END
00600C
00610C   D CALCULATES THE DERIVATIVE OF F
00620C
00630 FUNCTION D(X)
00640 IF(X.EQ.1.0) STOP "BAD ARG TO D"
00650 D=3.0+2.0/(X+1.0)**2
00660 RETURN
00670 END
```

Figure 3-25. Subroutine NEWT and Main Program Before Debugging

The next trap is an RJ trap, occurring when the function D is referenced. The PRINT command indicates that the input argument X0, has the correct value. An ABORT trap occurs at line 400, where the functon F is referenced. Since the function address was incorrectly specified in the CALL statement, F cannot be referenced in NEWT. However, an assignment command can be used to calculate the correct functional value and store it in FX. To continue execution, it is necessary to specify an address in the GO command to avoid executing the system

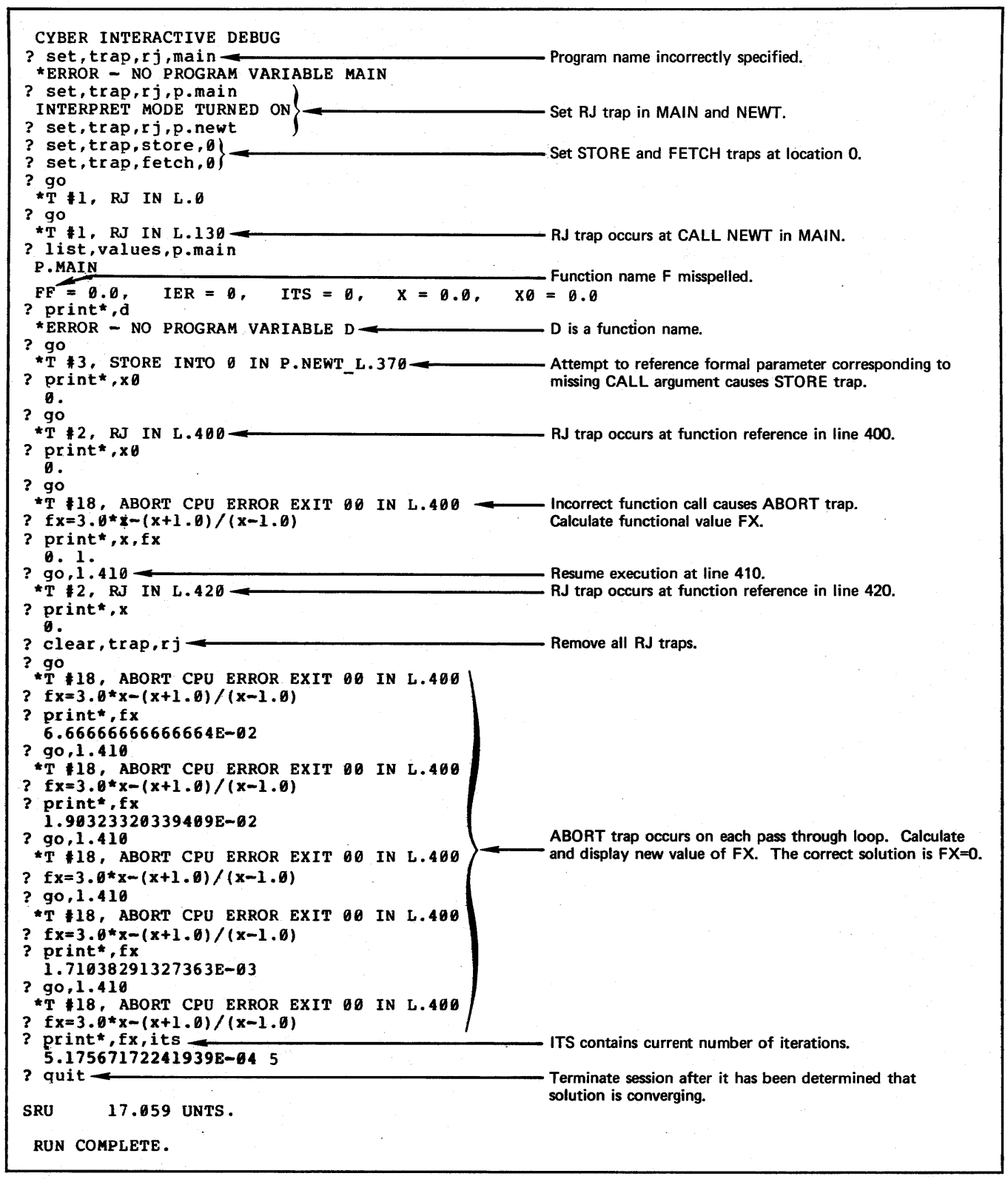

Figure 3-26. Debug Session for Subroutine NEWT

error code. In this case execution is resumed at line 410, the next statement. Note that in the case of an unresolved external reference, an RJ trap is not detected.

The RJ trap can now be removed with the CLEAR,TRAP command to avoid unnecessary interruptions, and CID still gains control through the ABORT trap, which occurs on each reference to the function F. Thus, on each pass though the loop, execution is suspended and a new functional value can be calculated with the assignment command.

If the iterative method is working properly, FX should approach zero. A few passes through the loop reveal that this is, in fact, happening. However, the test to exit from the loop is satisfied only if FX is equal to zero, and it is becoming evident that RX will approach but never equal zero.

To prevent an infinite loop, the test for convergence must be changed to exit on a sufficiently small value of FX. The constant used for the test depends on the desired degree of accuracy; for example, for 3-place accuracy, a value of .0001 would be used. A limit should also be imposed on the number of passes through the loop, since the method might not converge for certain functions. The loop can be replaced with a DO loop with an arbitrary limit of 100 passes. The corrected version of the program is shown in figure 3-27.

```
00100 PROGRAM MAIN(OUTPUT)
00110 EXTERNAL F,D
00120 X0=0.0
00130†CALL NEWT(F,D,X0,X,ITS,IER)
00140 IF(IER.NE.0) GO TO 900
00150 WRITE 100, ITS,X
00160 100 FORMAT(" CONVERGENCE IN ",I4," ITERATIONS.  X= ",E12.4)
00170 STOP
00180 900 STOP "ERROR IN SUBROUTINE NEWT"
00190 END
00200C
00210C   SUBROUTINE NEWT FINDS A ZERO ROOT OF AN EQUATION BY
00220C   NEWTONS METHOD
00230C
00240C   INPUT
00250C      F    NAME OF FUNCTION DEFINING EQUATION TO BE SOLVED
00260C      D    NAME OF FUNCTION DEFINING DERIVATIVE
00270C      X0   INITIAL APPROXIMATION TO ROOT
00280C
00290C   OUTPUT
00300C      X    SOLUTION TO F(X)=0
00310C      ITS  NUMBER OF ITERATIONS REQUIRED FOR SOLUTION
00320C      IER  ERROR FLAG
00330C           0  NO ERRORS
00340C           1  ERRORS
00350C
00360 SUBROUTINE NEWT(F,D,X0,X,ITS,IER)
00370 IER=0
00380 ITS=0
00385†EPS=0.0001
00390 X=X0
00391C
00392C    ITERATE TO FIND ROOT
00393C
00400†DO 10 I=1,100
00405 FX=F(X)
00410†IF(ABS(FX).LE.EPS) RETURN
00420 DX=D(X)
00430 IF(DX.EQ.0.0) GO TO 900
00440 X=X-FX/DX
00450 ITS=ITS+1
00460†10 CONTINUE
00462 WRITE 800
00464 800 FORMAT(" METHOD HAS NOT CONVERGED IN 100 ITERATIONS")
00465 IER=1
00466 RETURN
00467C
00470 900 WRITE 200, X0
00480 200 FORMAT(" DERIV 0 AT ",F6.2," SPECIFY DIFFERENT X0")
00490 IER=1
00500 RETURN
00510 END
00520C
00530C   F DEFINES A FUNCTION TO BE SOLVED BY NEWTONS METHOD
00540C
00550 FUNCTION F(X)
00560 IF(X.EQ.1.0) STOP "BAD ARG TO F"
00570 F=3.0*X-(X+1.0)/(X-1.0)
00580 RETURN
00590 END
00600C
00610C   D CALCULATES THE DERIVATIVE OF F
00620C
00630 FUNCTION D(X)
00640 IF(X.EQ.1.0) STOP "BAD ARG TO D"
00650 D=3.0+2.0/(X+1.0)**2
00660 RETURN
00670 END
```

†Indicates corrections.

Figure 3-27. Subroutine NEWT and Main Program With Corrections

This section describes some CID features and commands that allow you to obtain various kinds of information about the current debug session. These features include:

- Debug variables that contain useful information about the current session and which can be displayed at the terminal
- A HELP feature that provides information about CID commands
- LIST commands that can display such things as load map information, and trap and breakpoint information
- A TRACEBACK command that displays a subroutine traceback list
- Error and warning messages
- A command to control the types of output displayed at the terminal.
- A command to write CID output to an auxiliary file

The sample debug sessions appearing in this section, unless otherwise indicated, were produced by executing subroutine NEWT (figure 3-25) under CID control.

DEBUG VARIABLES

CID provides a set of variables that contain information about the current status of a debug session and of the executing program. You can display the contents of debug variables whenever you have control. CID updates these variables, and you cannot alter their contents directly.

Although the debug variables are primarily intended for use by assembly language programmers, some of the variables can provide information useful to FORTRAN programmers. Those variables that are most useful to FORTRAN programmers are listed in table 4-1. Refer to the CID reference manual for a description of all debug variables.

To display the contents of a debug variable, you must use the DISPLAY command; debug variables cannot be displayed with the PRINT command or LIST,VALUES command. All variables except #EW and #PC are automatically displayed in the appropriate format. Since #EW and #PC contain numeric values, you should specify the desired format on the DISPLAY command. Octal format is the default for these variables.

Examples:

DISPLAY,#EW,D
Display the value of #EW in decimal format.

DISPLAY,#LINE
Display the current source line number in the form P.name L.n.

TABLE 4-1. DEBUG VARIABLES

Variable	Description
#LINE	Number of FORTRAN line executing at time of suspension.
#EW	Effective word; on a STORE or FETCH trap, #EW is the value fetched or stored.
#PC	Previous contents; on STORE trap, #PC contains the value previously stored at the STORE location.
#EA	Effective address; depending on trap type, #EA indicates one of the following: STORE address (variable name) of store FETCH address (variable name) of fetch RJ address of program unit being called or location to which control is returned JUMP destination address of jump (S.n); undefined for conditional jumps if condition is false
#HOME	Home program name (P.name).
#BP	Number of breakpoints currently defined.
#TP	Number of traps currently defined.
#GP	Number of groups currently defined.

The #EW, #PC, and #EA variables are valid only when CID is in interpret mode. However, since the STORE, FETCH, RJ, and JUMP traps automatically activate interpret mode, it is not necessary to specify SET, INTERPRET when using the variables with these traps.

A debug session using debug variables is illustrated in figure 4-1. In this example traps and breakpoints are set in the main program and in subroutine NEWT. When execution is suspended, the DISPLAY command is used to display the values of various debug variables.

ERROR AND WARNING PROCESSING

Each time you enter a command, CID checks the command for correctness. If errors are detected, CID issues either an error or a warning message.

ERROR MESSAGES

CID issues an error message whenever it encounters a command that cannot be executed. Error messages are usually caused by a misspelled command or an illegal or misspelled parameter. CID does not attempt to execute an erroneous command. CID error messages, which are followed by a user prompt, have the form:

```
*ERROR-text
?
```

The text contains a brief description of the error.

In response to an error message, you should consult the CID reference manual or use the HELP command to determine the correct command form, and reenter the command.

Figure 4-2 shows some error messages that occurred when a SET,TRAP,STORE command was issued while debugging subroutine NEWT (figure 3-25).

```
 CYBER INTERACTIVE DEBUG
? set,trap,rj,p.main                          <- Set RJ trap in MAIN
 INTERPRET MODE TURNED ON
? set,breakpoint,p.newt_l.390                 <- Set breakpoint at line 390 of NEWT
? go
 *T #1, RJ IN L.0
? go
 *T #1, RJ IN L.130                           <- RJ trap at call to NEWT
? display,#ea                                 <- Display destination of call
 #EA = E.NEWT
? go
 *B #1, AT P.NEWT_L.390                       <- Breakpoint at line 390 of NEWT
? set,trap,store,x                            <- Set STORE trap for variable X
? display,#tp                                 <- Display number of traps currently defined
 #TP = 2
? display,#bp                                 <- Display number of breakpoints currently defined
 #BP = 1
? display,#line                               <- Display current FORTRAN line
 #LINE = P.NEWT_L.390
? display,#home                               <- Display home program name
 #HOME = P.NEWT
? go
 *T #2, STORE INTO X (OF P.MAIN) IN L.390     <- STORE trap on store into X at line 390
? display,#ea                                 <- Display address of store
 #EA = P.MAIN_X
? display,#ew,d                               <- Display value stored
 #EW = 0.0
? display,#pc,d                               <- Display previous contents of store location
 #PC = 0.0
? go
 *T #2, STORE INTO X (OF P.MAIN) IN L.440     <- STORE trap on store into X at line 440
? display,#ew,d                               <- Display value stored
 #EW = -.19999999999999928945726424
? display,#pc,d                               <- Display previous contents of store location
 #PC = 0.0
? quit

SRU      10.987 UNTS.

 RUN COMPLETE.
```

Figure 4-1. Debug Session Illustrating Debug Variables

```
? set,trap,store,p.newt_ex
 *ERROR - NO PROGRAM VARIABLE EX              <- Variable name incorrectly specified in preceding command.
? set,trap,store,p.newt_fx
 INTERPRET MODE TURNED ON
? set,breakpoint,s.10
 *ERROR - NO EXECUTABLE STATMENT 10           <- Qualifier omitted from statement label.
? set,breakpoint,p.newt_s.10
?
```

Figure 4-2. Partial Debug Session Illustrating Error Messages

WARNING MESSAGES

CID issues a warning message if a command you have entered will have consequences you might not be aware of or if the command will result in CID action other than that which you have specified. The warning message is followed by a special input prompt; in response to this prompt you can tell CID either to execute the command or to ignore it. The format of a warning message is:

```
*WARN-message
OK?
```

The message describes the action CID will take if allowed to execute the command. In response to a warning message you can enter the following:

YES or OK	CID executes the command.
NO	CID disregards the command.
Any CID Command	CID disregards the previous command and executes the new one.

Figure 4-3 illustrates some warning messages that were issued while debugging subroutine NEWT. In this example the programmer mistyped a breakpoint location and did not intend to redefine an existing breakpoint. CID notified the programmer of the effect and permitted the correct command to be entered.

Warning messages can be suppressed by an option on the SET,OUTPUT command, described later in this section. In this case, CID automatically takes the action indicated in the message without providing notification.

Refer to the CID reference manual for a complete list of warning messages and an explanation of each.

LIST COMMANDS

The LIST commands allow you to list various types of information relevant to the current debug session or to your program. The LIST commands are summarized in table 4-2.

The LIST commands are particularly useful with longer debug sessions in which you are constantly changing the status of the session. For example, you might initially set some traps or breakpoints, clear some or all of them later in the session, and set new ones; or you might change output options several times during the course of a session. With the LIST commands you can keep track of this and other CID information.

TABLE 4-2. LIST COMMANDS

Command	Description
LIST,BREAKPOINT	Lists breakpoint information.
LIST,TRAP	Lists trap information.
LIST,GROUP	Lists commands group information.
LIST,MAP	Lists load map information.
LIST,STATUS	Lists information about current status of debug session.
LIST,VALUES	Lists names and contents of user-defined variables.

Some of the LIST commands can produce a large volume of output. It is possible to prevent this output from appearing at the terminal and to write it instead to a separate file that can then be printed. The commands to accomplish this are discussed later in this section under Control of CID Output.

LIST,BREAKPOINT COMMAND

Breakpoint information can be displayed by issuing the LIST,BREAKPOINT command. The information included in the list depends on the form of the command used. You can list all breakpoints currently defined for your program or for a particular program unit or overlay. You can also specify individual breakpoints by address (line number or statement number) or by breakpoint number, in which case the breakpoint bodies, if any exist, are listed.

The following commands display a list of breakpoint locations:

LIST,BREAKPOINT
: Display a list of all breakpoints currently defined in the debug session.

LIST,BREAKPOINT,P.name
: Display a list of all breakpoints established in the specified program unit.

These forms of the LIST,BREAKPOINT command list the program unit in which the breakpoints reside and the line number (L.n) or statement label (S.n) of each breakpoint, depending on how you specified the breakpoint location in the SET,BREAKPOINT command. These commands do not list breakpoint bodies if any have been defined.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,p.newt_1.370
? set,breakpoint,p.newt_1.370,10,200,2   <-- Attempt to establish breakpoint where one already exists.
 *WARN - EXISTING BREAKPOINT WILL BE REDEFINED
 OK ? set,breakpoint,p.newt_s.10,10,200,2   <-- Enter correct command.
? clear,trap   <-- Attempt to clear all traps.
 *WARN - ALL WILL BE CLEARED
 OK ? ok   <-- Execute preceding command.
 INTERPRET MODE TURNED OFF
?
```

Figure 4-3. Partial Debug Session Illustrating Warning Messages

Breakpoint bodies (section 5) can be displayed by specifying individual breakpoints in the following formats:

LIST,BREAKPOINT,loc_1,loc_2,...
: Displays the bodies of the breakpoints at addresses loc_1, loc_2, ...; loc_n has the form L.n or S.n.

LIST,BREAKPOINT,#n_1,#n_2,...
: Displays the bodies of the breakpoints having sequence numbers #n_1, #n_2,...; breakpoint numbers are assigned by CID when the breakpoints are established.

Figure 4-4 illustrates the listing of breakpoints during a debug session for subroutine NEWT.

```
? list,breakpoint
 *B #1 = L.130,    *B #2 = P.NEWT_S.10
? clear,breakpoint
 *WARN - ALL WILL BE CLEARED
 OK ? ok
? list,breakpoint
 NO BREAKPOINTS
?
```

Figure 4-4. Partial Debug Session Illustrating LIST,BREAKPOINT Command

Examples of other forms of the LIST,BREAKPOINT command are included in section 5.

LIST,TRAP COMMAND

Information about traps defined for the current debug session can be displayed with the LIST,TRAP command. The information displayed depends on the form of the command used. The LIST,TRAP command has the following forms:

LIST,TRAP
: Display the type, address, and sequence number of all traps defined for the current session.

LIST,TRAP,type,P.name
: Display the address and sequence number of traps of the specified type defined for the specified program unit; type and P.name are optional.

LIST,TRAP,type
: Display the address and sequence number of traps of the specified type; type is one of the types listed in table 3-1.

LIST,TRAP,,P.name
: Display the address and sequence number of all traps defined for the specified program unit.

Following is an example of LIST,TRAP output:

```
T #1 = RJ P.TEST,   T #2 = RJ P.SUBX,
 T #3 = STORE X,    T #4 = FETCH X
```

Trap #1 is an RJ trap established in program unit TEST; trap #2 is an RJ trap established in program unit SUBX; trap #3 is a STORE trap for the variable X; and trap #4 is a FETCH trap for X.

The preceding LIST,TRAP commands do not list trap bodies if any have been defined. To list the commands comprising trap bodies, use the form:

LIST,TRAP,#n_1,#n_2,...

where #n_1,#n_2,... are trap sequence numbers assigned by CID when the trap is established. Trap bodies are discussed in section 5.

For all LIST,TRAP commands, CID lists the trap location in the same manner it was specified in the SET,TRAP command. This location can be an entire program unit (P.name), a line number (L.n), a statement number (S.n), a qualified line or statement number (P.name_L.n or P.name_S.n), or a variable name.

Figure 4-5 illustrates LIST,TRAP commands used in a debug session for subroutine NEWT. After the first LIST,TRAP is issued, all traps are removed with the CLEAR,TRAP command and another LIST,TRAP is issued.

LIST,GROUP COMMAND

Information about command groups defined for the current session can be displayed with the LIST,GROUP command. This command has the following forms:

LIST,GROUP
: List the names and numbers of all groups defined for the current session.

LIST,GROUP,$name_1$,$name_2$,...
: List the commands contained in the groups having the specified names.

LIST,GROUP,#n_1,#n_2,...
: List the commands contained in the groups identified by the specified numbers.

Note that the first command form lists only the names and numbers of groups, whereas the second and third forms list the commands comprising the specified groups.

Groups are discussed in section 5.

```
? list,trap
 T #1 = STORE P.NEWT_FX,    T #2 = STORE P.NEWT_DX
? clear,trap,#1,#2
 INTERPRET MODE TURNED OFF
? list,trap
 NO TRAPS
?
```

Figure 4-5. Partial Debug Session Illustrating LIST,TRAP Command

LIST,MAP COMMAND

The LIST,MAP command displays load map information. This command is useful when the FORTRAN program contains many subroutine calls or common blocks, since it provides a concise list of subroutine and common block names. The LIST,MAP command can also provide length information which is useful in detecting incorrectly specified common block lengths. This command has the following forms:

LIST,MAP
List all modules comprising the program, including user-defined modules, common blocks, and FORTRAN and library modules.

LIST,MAP,P.name$_1$,P.name$_2$,...
List the first word address (FWA), length (octal words), and all entry point names for the specified program units.

LIST,MAP,C.name$_1$,C.name$_2$,...
List the first word address and length (octal words) of the specified common blocks.

Common blocks are enclosed in slashes in LIST,MAP output.

Figure 4-6 illustrates a debug session involving a program in which two common blocks are declared. An incorrect dimension is specified for common block ACOM in subroutine BAKER. The LIST,MAP command displays the correct length as declared in the main program.

LIST,STATUS COMMAND

The LIST,STATUS command displays a brief summary of the status of a debug session as it exists at the time the command is issued. This command has the form:

LIST,STATUS

Sample Program:

```
      PROGRAM ABLE        74/74

1                 PROGRAM ABLE
                  COMMON/ACOM/A(10),AA(10)
                  COMMON/BCOM/B(50),BB(100)
                  CALL BAKER
5                 STOP
                  END


 SUBROUTINE BAKER       74/74

1                 SUBROUTINE BAKER
                  COMMON/ACOM/X(25)
                  COMMON/BCOM/Y(1)
                  DO 6 I=1,25
5           6     X(I)=0.0
                  DO 8 I=1,150
            8     Y(I)=0.0
                  RETURN
                  END
```

Session Log:

```
CYBER INTERACTIVE DEBUG

list,map
                    user programs
DBUG.,   ABLE,   /ACOM/,   /BCOM/,   BAKER,   Q2NTRY=,   /STP.END/
/FCL.C./,   /Q8.IO./,   FCL=FDL,   /FCL=ENT/,   FEIFST=,   FORSYS=
GETFIT=,   SYSAID=,   FORUTL=,   FTNRP2=,   UCLOAD,   FDL.RES
/FDL.COM/,   FDL.MMI,   CPU.SYS,   CMF.ALF,   CMF.CSF,   CMM.FFA
CMF.FRF,   CMM.R,   CMF.SLF,   CTL$RM,   ERR$RM,   LIST$RM
?list,map,c.acom <-- Display starting address and length of ACOM.

BLOCK - ACOM,   FWA = 3121B,   LENGTH = 24B
?list,map,c.bcom <-- Display starting address and length of BCOM.

BLOCK - BCOM,   FWA = 3145B,   LENGTH = 226B
?quit

  DEBUG TERMINATED
```

Figure 4-6. Program ABLE and Debug Session Illustrating LIST,MAP Command

Information displayed by the LIST,STATUS command includes:

Home program name.

Number of breakpoints currently defined.

Number of traps currently defined.

Number of groups currently defined.

Veto mode on or off.

Interpret mode on or off.

Output options. Output options are controlled by the SET,OUTPUT command, described under Control of CID Output, which specifies the types of CID output sent to the terminal.

Auxiliary file options. These options are specified by the SET,AUXILIARY command (section 5), which defines an auxiliary output file and specifies the type of output to be sent to that file.

Figure 4-7 illustrates LIST,STATUS commands issued when debugging subroutine NEWT. The command is issued at the beginning of the session and again when execution is suspended at a breakpoint in NEWT. The output indicates the changes in the status of the debug session.

HELP COMMAND

CID provides a HELP command that displays a brief summary of information about specific CID subjects and commands. You can issue the HELP command whenever you need assistance with a particular aspect of CID.

Simply entering the command:

HELP

causes CID to display a list of subjects. To obtain additional information about any subject in the list, enter:

HELP,subject

For example, the command HELP,ERROR displays a brief description of error processing.

A particularly useful command is HELP,CMDS which displays a complete list of CID commands and a brief explanation of each. You can obtain a more detailed explanation of any CID command by entering:

HELP,command

The HELP command does not provide the same level of detail as the CID reference manual, however, and should not be considered a substitute for the reference manual.

The HELP command is illustrated in figure 4-8, which shows the entry of the command HELP,SET,BREAKPOINT to display a summary of the command parameters.

TRACEBACK COMMAND

The TRACEBACK command displays a list of subroutine levels from the level of the most recent execution of the specified subprogram through the main level. At each level, TRACEBACK displays the name of the program unit that last called the specified subroutine and the line number within the program unit where the call occurred. The format of this output is P.name L.n. The TRACEBACK command has the following forms:

TRACEBACK
Display a traceback beginning at the current home program.

TRACEBACK,P.name
Display a traceback beginning with the specified program unit name.

TRACEBACK,E.ept
Display a traceback beginning with the specified entry point name.

The TRACEBACK command is illustrated by the debug session shown in figure 4-9. In this example, breakpoints are established in function subprograms D and F. When execution is suspended in these functions, various forms of the TRACEBACK command are issued.

CONTROL OF CID OUTPUT

The output produced by the LIST, TRACEBACK, DISPLAY, and PRINT commands can become voluminous. As an alternative to displaying all CID output at the terminal, CID provides the following commands for controlling the disposition of output:

```
?list,status

HOME = P.MAIN,   NO BREAKPOINTS,   NO TRAPS,   NO GROUPS,   VETO OFF
INTERPRET OFF,   OUT OPTIONS = I W E D,   AUXILIARY CLEAR
?

                                   •
                                   •
                                   •

?list,status

HOME = P.NEWT,   2 BREAKPOINTS,   1 TRAPS,   NO GROUPS,   VETO OFF
INTERPRET ON,   OUT OPTIONS = I W E D,   AUXILIARY CLEAR
?
```

Figure 4-7. Partial Debug Session Illustrating LIST,STATUS Command

```
 CYBER INTERACTIVE DEBUG
? help,set,breakpoint
 SB  -  SET  BREAKPOINT - ALLOWS YOU TO SET A BREAKPOINT AT A
 SPECIFIC LOCATIONS IN USER'S PROGRAM.  THE FORM OF  THE  SET
 BREAKPOINT COMMAND IS.
 SB LOCATION, FIRST, LAST, STEP
 WHERE  'LOCATION'  IS  THE LOCATION IN YOUR PROGRAM AT WHICH
 YOU WANT THE BREAKPOINT SET.
 'FIRST', 'LAST' AND 'STEP' ARE OPTIONAL AND ARE DEFAULTED TO
 1, 131071 AND 1 RESPECTIVELY.  THE BREAKPOINT IS NOT HONORED
 UNTIL IT HAS BEEN HIT 'FIRST' TIMES, WHEN IT WILL BE HONORED
 EACH 'STEP'TH TIME UNTIL 'LAST' IS REACHED OR EXCEEDED.
 IF YOU TERMINATE THE SB COMMAND WITH AN OPEN BRACKET [, THEN
 ALL COMMANDS UP TO A CLOSE BRACKET ] WILL BE COLLECTED  SUCH
 THAT  WHEN THE BREAKPOINT IS HONORED, THOSE COMMANDS WILL BE
 EXECUTED.
?
```

Figure 4-8. Partial Debug Session Illustrating HELP Command

```
? set,breakpoint,p.f_1.580
? set,breakpoint,p.d_1.660
? go
 *B #1, AT P.F_L.580 <-------- Execution suspended in function F.
? traceback <---------------- Initiate traceback form home program.
 P.F CALLED FROM P.NEWT_L.0
 P.NEWT CALLED FROM P.MAIN_L.130
? traceback,p.d <------------- Attempt to initiate traceback from a subprogram that has
 *ERROR - PROGRAM D NOT CALLED   not been executed.
? go
 *B #2, AT P.D_L.660 <-------- Execution suspended in function D.
? traceback <---------------- Initiate traceback from home program.
 P.D CALLED FROM P.NEWT_L.0
 P.NEWT CALLED FROM P.MAIN L.130
? traceback,p.f <------------- Initiate traceback from function F.
 P.F CALLED FROM P.NEWT_L.0
 P.NEWT CALLED FROM P.MAIN L.130
?
```

Figure 4-9. Debug Session Illustrating TRACEBACK Command

SET,OUTPUT
: Specify the types of output to be displayed at the terminal.

SET,AUXILIARY
: Define an auxiliary output file and specifies the types of output to be sent to the file.

TYPES OF OUTPUT

For purposes of the SET,OUTPUT and SET,AUXILIARY commands, CID output is classified as to type with each type represented by a 1-letter code. The output codes, along with a description of each, are listed in table 4-3.

SET,OUTPUT COMMAND

The SET,OUTPUT command specifies the types of output to be displayed at the terminal. The SET,OUTPUT command has the form:

SET,OUTPUT,t_1,t_2,...

where t_n is an optional output code. Valid codes are listed in table 4-3.

Including an output code in the option list of the SET,OUTPUT command causes the associated output type to be displayed at the terminal. Omitting an output code from the option list suppresses the associated output type. Thus, when a SET,OUTPUT command is specified, any output type not included in the option list is not displayed at the terminal. For example, the command:

SET,OUTPUT,E,W,I

causes output types E, W, and I to be displayed at the terminal while suppressing types D, R, and B.

The default output types are E, W, D, and I. It is unnecessary to specify type T in a SET,OUTPUT command since all user input is displayed at the terminal when it is entered.

The only output types not automatically displayed are group and file command sequences (type R) and trap and breakpoint bodies (type B). To display this output, in addition to the default types, enter the command:

SET,OUTPUT,E,W,I,D,R,B

TABLE 4-3. CID OUTPUT TYPES

Output Code	Description
E	Error messages.
W	Warning messages.
D	Output produced by execution of CID commands. Includes output resulting from LIST, DISPLAY, PRINT, TRACEBACK commands.
I	Informative messages. Includes trap and breakpoint messages and mode messages.
R	Group and file command sequences; output when a READ command is executed.
B	Trap and breakpoint body command sequences; output when a trap or breakpoint with a body is encountered.
T	Echo of user-input information.

If you specify the R option on the SET,OUTPUT command, then whenever a READ command is executed, the command sequence is displayed at the terminal. If you specify the B option, then whenever a trap or breakpoint for which you have defined a body is detected, the commands comprising the body are displayed. Command sequences are discussed in section 5.

The only output types that cannot be suppressed are the informative messages issued when traps or breakpoints are detected (these are included in type I). These messages are always displayed regardless of SET,OUTPUT specifications. Error messages (type E) can be suppressed only if you have provided for writing them to an auxiliary file with the SET,AUXILIARY command. If you attempt to suppress error messages and you have not provided for writing them to an auxiliary file, CID issues an error message.

If you suppress warning messages by omitting W from the SET,OUTPUT command, CID executes all commands that would normally generate a warning message. No user prompt is issued; CID takes the corrective action described in the warning message, as if you had entered a YES or OK response (see Error and Warning Processing).

To suppress all output to the terminal (except trap and breakpoint messages) you can issue either a SET,OUTPUT command with no option list or the command:

```
CLEAR,OUTPUT
```

Prior to entering either of these commands, however, you must provide for writing error messages to an auxiliary file.

After a CLEAR,OUTPUT command has been issued, you can restore output to default conditions with the command:

```
SET,OUTPUT,E,W,D,I
```

The SET,OUTPUT command can be used in conjunction with the SET,AUXILIARY command to suppress certain types of output to the terminal and to send that output type to an auxiliary file. The most common output to suppress is type D, output produced by execution of CID commands. This includes output produced by the LIST and display commands, all of which can produce large amounts of output.

SET,AUXILIARY COMMAND

The SET,AUXILIARY command defines an auxiliary output file and specifies which types of CID output are to be written to that file. The SET,AUXILIARY command has the following form:

```
SET,AUXILIARY,lfn,t1,t2,...
```

where lfn is the name of the auxiliary file and t_n is an output code. Valid codes are listed in table 4-3.

The SET,AUXILIARY command has no effect on output that is being displayed at the terminal. For example, the command:

```
SET,AUXILIARY,FAUX,I,D
```

creates a file named FAUX and writes all informative and command output messages to the file. These messages are also displayed at the terminal unless the appropriate SET,OUTPUT command has been used to change this.

The option specifications for an auxiliary file can be changed simply by entering another SET,AUXILIARY command with the file name and a new option list; it is not necessary to close the file beforehand.

Only one auxiliary file can be in use at a time. The QUIT command closes the auxiliary file currently in use. To close an auxiliary file before the end of a debug session, issue the command:

```
CLEAR,AUXILIARY
```

An auxiliary file can be closed at any time during a debug session.

After you close an auxiliary file you can dispose of it in any manner you wish, displaying it at the terminal, sending it to a printer, or storing it on a permanent storage device. CLEAR,AUXILIARY does not rewind the file; after issuing a CLEAR,AUXILIARY you can issue a SET,AUXILIARY for the same file in the same or in a subsequent session and the additional information is written after the end-of-record.

A common use of the SET,AUXILIARY command is to preserve a copy of a debug session log. The command:

```
SET,AUXILIARY,OUTF,E,W,D,I,T
```

issued at the beginning of a debug session writes to file OUTF the output types E, W, D, I, and T, thus creating a copy of the session exactly as displayed at the terminal. Note that user commands automatically appear when outputting to the terminal. However, when outputting to an auxiliary file, you must specify the T option to include user-entered commands in the file.

The following example illustrates a SET,OUTPUT command used in conjunction with a SET,AUXILIARY command to suppress output to the terminal and write it to an auxiliary file:

```
?SET,OUTPUT,E,W,I
?SET,AUXILIARY,LFG,D
?LIST,MAP
?CLEAR,AUXILIARY
?SET,OUTPUT,E,W,I,D
```

This example suppresses all output produced by CID commands (type D), creates an auxiliary file called LGF to which this output is to be written, writes load map information to LGF, closes LGF, and resets output options to original conditions.

The following example illustrates a CLEAR,OUTPUT command used with a SET,AUXILIARY command:

```
?SET,AUXILIARY,AUXF,D,E
?CLEAR,OUTPUT
?LIST,VALUES
?CLEAR,AUXILIARY
?SET,OUTPUT,E,W,D,I
```

This example defines an auxiliary file named AUXF to receive error messages and output from CID commands, turns off output to the terminal (except for trap and breakpoint messages), writes program variables and contents to AUXF, closes AUXF, and restores terminal output to normal default conditions.

An example of a debug session using an auxiliary file is illustrated in figure 4-10. This session was produced by executing subroutine AREA (figure 3-10) under CID control. In this example, an auxiliary file AFILE is defined; the D option causes output from CID commands to be sent to AFILE. A breakpoint is established at line 6. After the breakpoint suspends execution, commands are issued to suppress command output (type D) to the terminal, to list all variables and values local to subroutine AREA, and then to reestablish default output conditions. The session is terminated after one pass through AREA. File AFILE (figure 4-11) contains the output from the LIST,VALUES command. (A better way of doing this would be to include the SET,OUTPUT and LIST,VALUES commands in a breakpoint body. This would preclude the necessity of reentering these commands on each pass through the subroutine. Breakpoint bodies are discussed in section 5.)

```
CYBER INTERACTIVE DEBUG
?set,auxiliary,afile,d,e <-- Establish auxiliary file AFILE. Command output and error messages
                             are to be sent to this file.
?set,breakpoint,p.area_1.7

?go

*B #1, AT P.AREA_L.7
?clear,output <-- Clear all output to terminal.

?list,values,p.area

?go

*B #1, AT P.AREA_L.7
?list,values,p.area <-- List program values while execution is suspended on each pass
                        through subroutine AREA.
?go

*B #1, AT P.AREA_L.7
?list,values,p.area

?set,output,e,w,d,i <-- Restore normal output to terminal.

?go

*T #17, END IN P.RDTR_L.7
?
     STOP
      1.406 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Figure 4-10. Debug Session Illustrating SET,AUXILIARY, SET,OUTPUT and CLEAR, OUTPUT Commands

```
1                                                    CYBER INTERACTIVE DEBUG
0
 *B #1, AT P.AREA_L.7
 P.AREA
 A = .54999999999999,    S1 = 1.1180339887499,    S2 = 1.0198039027186 }<-- First pass
 S3 = 1.7,    T = 1.9189189457342,    X1 = 0.0,    X2 = .5,    X3 = -1.0 }
 Y1 = 1.0,    Y2 = 2.0,    Y3 = 1.2
 *B #1, AT P.AREA_L.7
 P.AREA
 A = 23.699999999999,    S1 = 8.4852813742385,    S2 = 5.7801384066474 }<-- Second pass
 S3 = 11.428473213864,    T = 12.846946497375,    X1 = 6.1,    X2 = .1 }
 X3 = 3.2,    Y1 = 2.0,    Y2 = -4.0,    Y3 = 7.0
 *B #1, AT P.AREA_L.7
 P.AREA
 A = 33.704999999999,    S1 = 11.002726934719,    S2 = 7.8517513969814 }<-- Third pass
 S3 = 8.6400231481171,    T = 13.747250739909,    X1 = .2,    X2 = -1.3 }
 X3 = 5.6,    Y1 = -2.9,    Y2 = 8.0,    Y3 = 2.8
 *T #17, END IN P.RDTR_L.7
```

Figure 4-11. Listing of Auxiliary File AFILE

AUTOMATIC EXECUTION OF CID COMMANDS 5

It is frequently necessary to enter the same command or sequence of commands many times during the course of a debug session. This situation is illustrated in the examples in figures 3-16 through 3-27. In the debug session involving subroutine NEWT, the same sequence of commands is needed on each pass through the loop. Debugging program CORR required several sessions; during each session it was necessary to reenter the assignment commands that calculated the correct intermediate values.

To eliminate the need for repeatedly entering sequences of commands, CID provides the ability to define, save, and automatically execute sequences of commands. Sequences can be used to improve debugging efficiency whenever the same group of CID commands must be entered repeatedly. Sequences are commonly used when debugging DO loops and frequently called subroutines, and in multiple debug sessions that require the same commands. In addition, CID provides some special sequence commands that allow you to incorporate FORTRAN-like logic into command sequences. For example, sequence commands allow branching and conditional execution of CID commands.

COMMAND SEQUENCES

A command sequence is a series of CID commands which is to be executed automatically either when certain conditions occur or when the appropriate command is entered from the terminal.

There are three ways in which you can establish a command sequence:

- By defining a command sequence as part of a trap or breakpoint. This causes the sequence to be executed whenever the trap or breakpoint occurs. A sequence defined in this manner is called a trap body or breakpoint body.
- By defining a command sequence called a group. A group can be executed by issuing a READ command from the terminal or from another command sequence.
- By creating a file, outside of CID, which contains a sequence of CID commands. The commands in this file can be executed by issuing a READ command at the terminal or from another command sequence.

During normal execution, CID prompts for user input after a command is executed. During sequence execution, however, CID executes all the commands in the sequence without interruption. Once execution of the sequence is completed, execution of your program resumes at the point where it was suspended. The PAUSE command, described later in this section, allows you to interrupt the execution of a command sequence.

Command sequences can be nested; that is, command sequences can be called from other command sequences.

COLLECT MODE

Collect mode is a mode of execution in which CID commands are not executed immediately, but are included in a command sequence for execution at a later time. To define a trap body, breakpoint body, or command group, you must first activate collect mode. The procedure for entering and leaving collect mode is described under Traps and Breakpoints With Bodies.

Commands in a sequence that you are creating cannot be altered while CID is in collect mode. If you make a mistake or wish to change a command that you have entered, you must leave collect mode and proceed as described under Editing a Command Sequence.

SEQUENCE COMMANDS

The commands intended specifically for use with command sequences are summarized in table 5-1.

TABLE 5-1. SEQUENCE COMMANDS

Command	Description
PAUSE	Temporarily suspends execution of the current command sequence and reinstates interactive mode allowing commands to be entered from the terminal.
MESSAGE	Displays a message at the terminal.
GO	Resumes the process most recently suspended.
EXECUTE	Resumes execution of the user program.
IF	Performs conditional execution of commands.
LABEL	Defines a label within a command sequence.
JUMP	Transfers control within a command sequence to a label defined by a LABEL command.
READ	Initiates execution of a command sequence defined as a group or stored on a file; reestablishes trap, break-point, and group definitions stored on a file.

TRAPS AND BREAKPOINTS WITH BODIES

A body is a sequence of commands specified as part of a SET,TRAP or SET,BREAKPOINT command. To define a trap or breakpoint with a body, you must first initiate collect mode by including a left bracket ([) as the last parameter of the SET,TRAP or SET,BREAKPOINT command. For example:

```
SET,TRAP,LINE,P.MAIN [
```

The bracket and the preceding parameter must not be separated by a comma; the blank separator is optional.

When the above command is entered, CID displays the message and prompt:

```
IN COLLECT MODE
?
```

You then enter the commands that are to comprise the body. Each command entered while CID is in collect mode becomes part of the body. CID scans the command for errors but does not execute the command. You can include any number of commands in a body, although command sequences should be kept short and simple.

To leave collect mode and return to interactive mode, enter a right bracket ( ) in response to the ? prompt or at the end of a command line. CID then displays:

```
END COLLECT MODE
?
```

and you can continue the session.

An example of a breakpoint definition with a body is as follows:

```
SET,BREAKPOINT,L.8, [
A=B-C
X=0.0
Y=Y+1.0
PRINT*,A,X,Y
]
```

When a trap or breakpoint with a body is encountered, program execution is suspended and the commands in the body are executed automatically. Program execution then resumes at the trap or breakpoint location.

When a trap or breakpoint with a body is encountered during execution, the normal trap or breakpoint message is not displayed. However, you can provide your own notification of the execution of a trap or breakpoint body by including a MESSAGE command (table 5-1) in the sequence. The format of the MESSAGE command is:

```
MESSAGE,"character string"
```

When a MESSAGE command is encountered, the character string is displayed.

You do not receive control during execution of a sequence unless you have provided for it by including a PAUSE command (described under Receiving Control During Sequence Execution) in the body. When the body has been executed, execution of your program automatically resumes at the location where it was suspended.

You can list the commands contained in a trap body by issuing a LIST,TRAP command and specifying a list of trap numbers, as in the example:

```
LIST,TRAP,#2,#3
```

Other forms of the LIST,TRAP command list the type and location of the trap, but not the body.

To list the commands contained in a breakpoint body, specify the breakpoints either by number or location, as in the examples:

```
LIST,BREAKPOINT,#1,#5,#6
LIST,BREAKPOINT,P.SUBC_L.10
```

An example of the procedure for establishing a breakpoint body is illustrated in figure 5-1. The program used in this example is shown in figure 3-1. A breakpoint is established at the RETURN statement in subroutine AREA. The breakpoint body contains the following commands:

A MESSAGE command to display a message when the trap occurs

A DISPLAY command to display the contents of the #LINE variable which contains the current line number

A PRINT command to display the input values and the value of A

After the trap is established, the LIST,GROUP command displays the commands comprising the trap body.

Subroutine AREA is called three times; each time the breakpoint is detected, the commands in the sequence are executed.

GROUPS

A group is a sequence of commands established and assigned a name during a debug session, but not explicitly associated with a trap or breakpoint. A group exists for the duration of the session and can be executed by entering an appropriate READ command. The command to establish a group is:

```
SET,GROUP,name [
```

where name is a name by which you will reference the group. The left bracket activates collect mode, as with trap and breakpoint bodies. Any number of CID commands subsequently entered become part of the sequence until you terminate the sequence by entering a right bracket.

To execute a group issue the command:

```
READ,name
```

where name is the group name assigned in the SET,GROUP command. You can issue a READ command directly from the terminal while CID is in interactive mode, or from another command sequence. In response to a READ command, CID executes the commands in the group. After a group has been executed, control automatically returns to CID if the READ was entered from the terminal, or to the next command in the sequence that issued the READ.

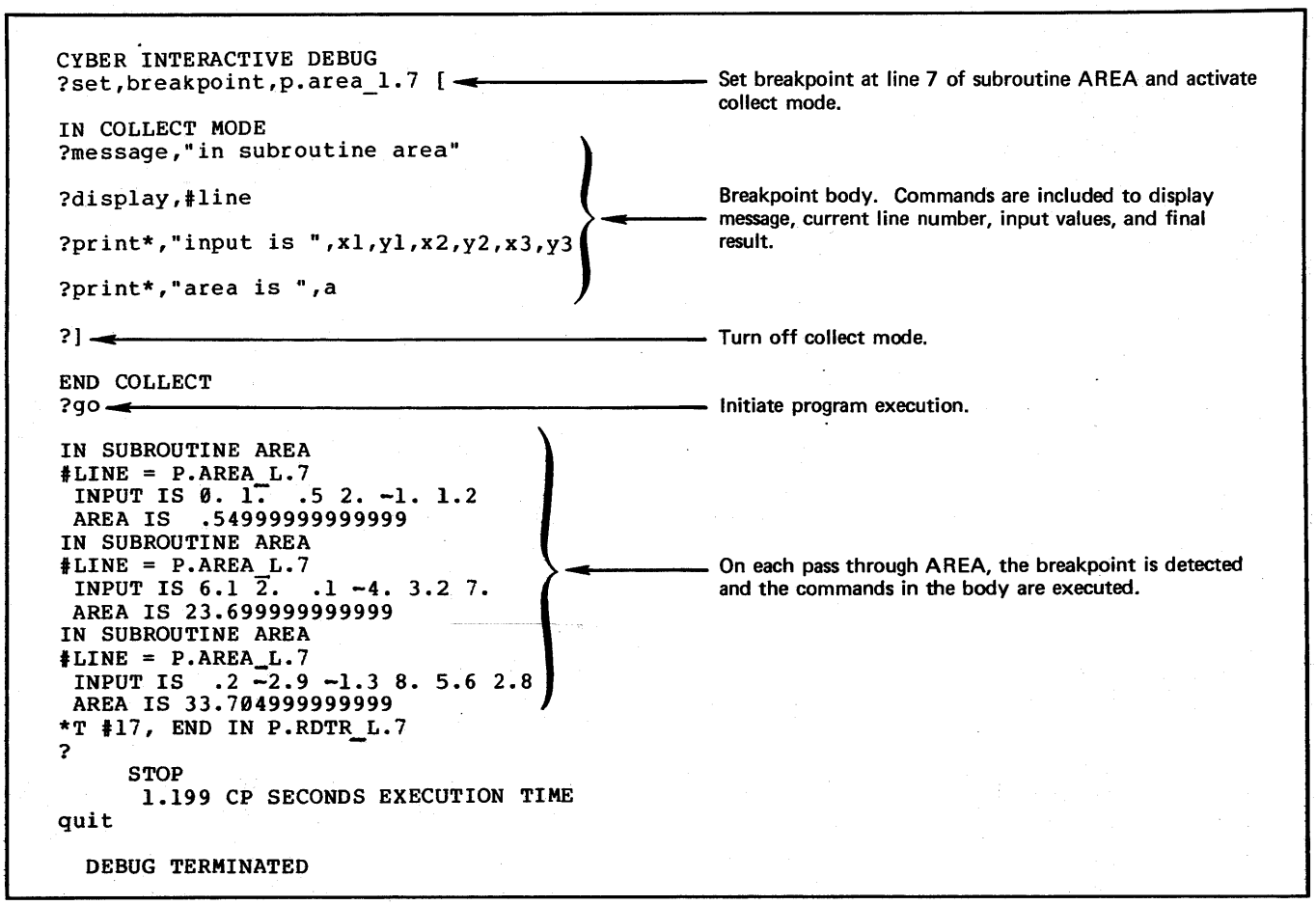

Figure 5-1. Debug Session Illustrating Breakpoint With Body

A group can be used when the same sequence of commands is to be executed at different locations in a program. A trap or breakpoint body is executed only when the trap or breakpoint occurs, but a group can be executed at any time. A READ command can be included in any command sequence, allowing nesting of groups. Following is an example of a simple group definition:

```
SET,GROUP,GRPA[
X=Y+Z
PRINT*,X,Y,Z
]
```

This command sequence can be executed by issuing the command:

```
READ,GRPA
```

When a group is established, it is assigned a number in the same manner as traps and breakpoints. You can refer to a group by number or by name in the LIST, CLEAR, and SAVE commands.

You can list the commands comprising a group with the following commands:

LIST,GROUP
: List the names and numbers of all groups defined for the current debug session; does not list the commands contained in the groups.

LIST,GROUP,$name_1$,$name_2$,...
: List the commands contained in the specified groups.

LIST,GROUP,#n_1,#n_2,...
: List the commands contained in the groups identified by the specified numbers.

Normally, a group exists for the duration of a debug session. You can remove a group or groups from the current debug session by entering one of the following commands:

CLEAR,GROUP
: Remove all currently defined groups.

CLEAR,GROUP,$name_1$,$name_2$,...
: Remove the specified groups.

CLEAR,GROUP,#n_1,#n_2,...
: Remove the groups identified by the specified numbers.

Figures 5-2 and 5-3 illustrate debug sessions using groups. In figure 5-2, two breakpoints are set in subroutine SETB. When either breakpoint is reached, the READ command is issued from the terminal. In figure 5-3, the same breakpoints are established, except that a body containing a READ command is defined for each. This causes the body to be executed automatically when the breakpoints

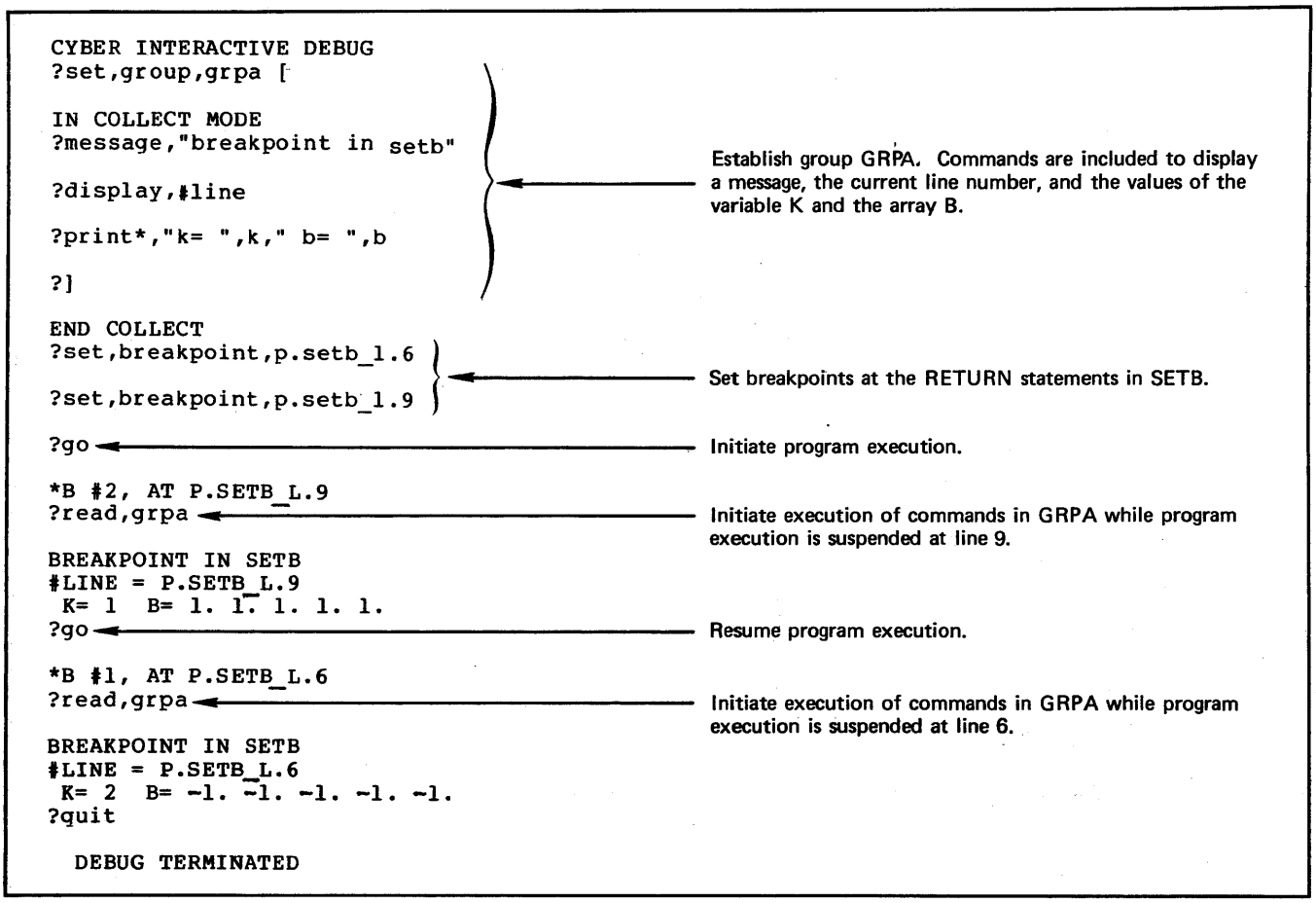

Figure 5-2. Debug Session Illustrating Group Execution Initiated at the Terminal

are encountered, with no intervention from the user. By defining a single group, instead of a body for each breakpoint, it is necessary to enter the command sequence only once. The group is listed with the LIST,GROUP command.

In figure 5-3, note that there are three levels of execution: the program, the breakpoint body, and the group. When the breakpoint is reached, the program is suspended and execution of the breakpoint body is initiated. When the READ command is encountered, execution of the breakpoint body is suspended while the group is executed. When execution of the group is complete, execution of the suspended breakpoint body resumes at the command following the READ. When execution of the breakpoint body is complete, execution of the suspended program resumes.

Groups are especially useful when the same sequence of commands is to be executed at more than one location within a program. An example of this is illustrated in figure 5-4. The program MATOP defines two matrices and calls subroutines to add, subtract, and multiply the matrices and store the results in a work area called MWRK. The purpose of the debug session is to print the contents of MWRK after each subroutine call. To accomplish this a group named PRNT is established containing appropriate PRINT commands. After each subroutine call, a breakpoint is set with a body containing a command to execute the commands in group PRNT. When each breakpoint is encountered, the group commands are automatically read and executed. The debug session in figure 5-5 is identical except that the command READ,PRNT is issued from the terminal instead of a breakpoint body.

ERROR PROCESSING DURING SEQUENCE EXECUTION

When CID is in collect mode and you are defining a command sequence, CID scans each command you enter for syntactic errors. If a syntactic error is detected, CID displays an error message and ? prompt, after which you can reenter the command. Other errors, however, such as nonexistent line number or variable name, cannot be detected until CID attempts to execute the command.

CID issues normal error and warning messages during sequence execution. When an error or warning condition is detected, CID suspends execution of the sequence and issues a message followed by an input prompt (? for error messages; OK? for warning messages) on the next line. You can instruct CID to disregard the command, replace the command with another command, or, in the case of warning messages, execute the command. The most

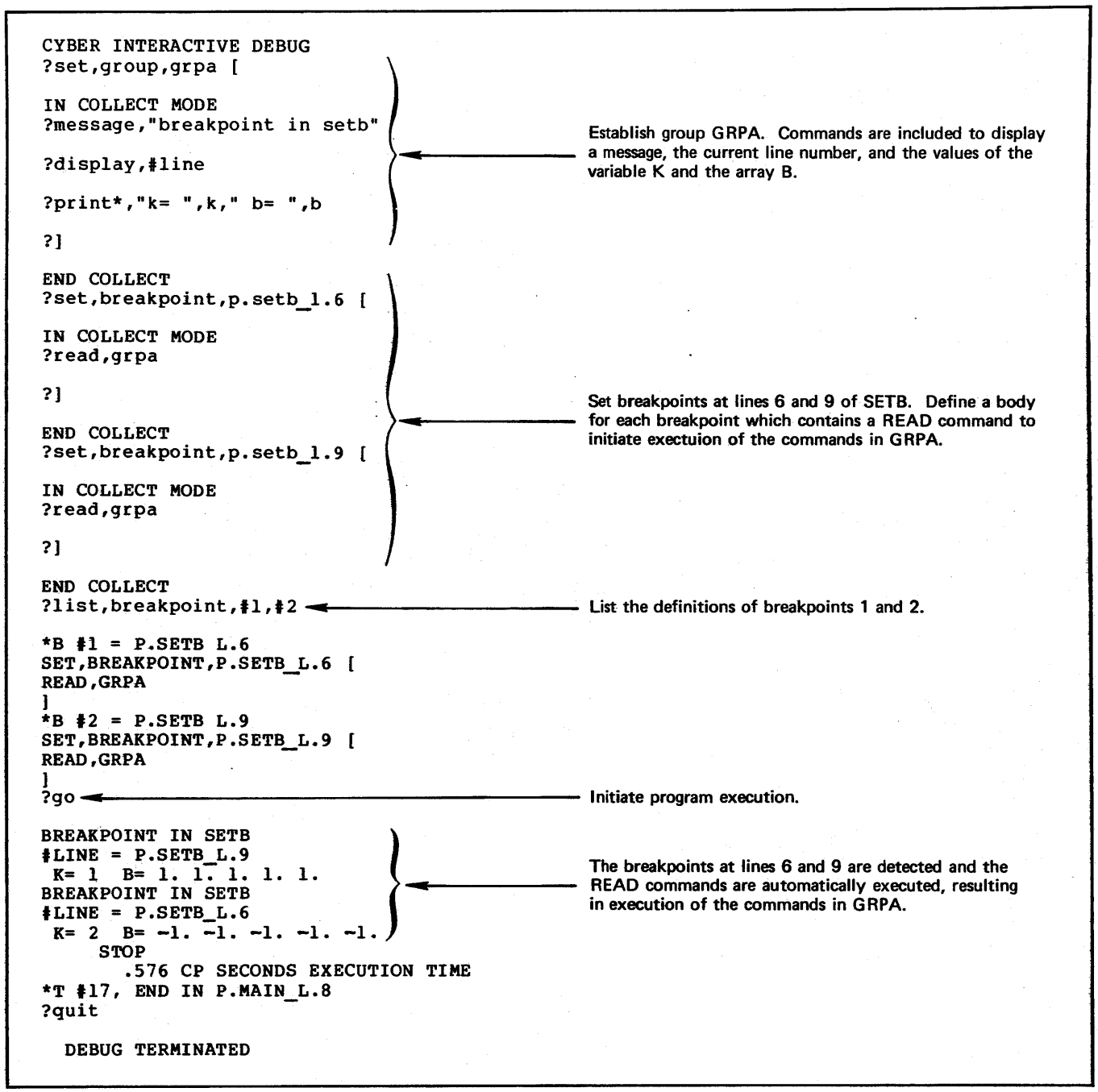

Figure 5-3. Debug Session Illustrating Group Execution Initiated From Breakpoint Body

useful ways in which you can respond to error and warning messages are summarized as follows:

User Response	Debug Action
OK or YES	For warning messages only, execute the command.
NO	Disregard the command. Execution resumes at the next command in the sequence.
NO,SEQ	Disregard the command and all remaining commands in the sequence.
Any CID command	Execute the specified command line in place of the current command and resume execution of the sequence.

Refer to the CYBER Interactive Debug reference manual for other valid responses to error and warning messages.

Program MATOP:

```
1             PROGRAM MATOP      74/74   TS ID


                           PROGRAM MATOP
                           DIMENSION MAT1(3,3),MAT2(3,3),MWRK(3,3)
                           DATA MAT1/2,6,4,3,8,9,7,5,8/
                          1,    MAT2/1,0,0,0,1,0,0,0,1/
         5                 N=3
                           CALL MATADD(N,MAT1,MAT2,MWRK)
                           CALL MATSUB(N,MAT1,MAT2,MWRK)
                           CALL MATMPY(N,MAT1,MAT2,MWRK)
                           STOP
        10                 END
```

Session Log:

```
CYBER INTERACTIVE DEBUG
?set,group,prnt [

IN COLLECT MODE
?message,"contents of mwrk"

?display,#line

?print*,mwrk(1,1),mwrk(1,2),mwrk(1,3)

?print*,mwrk(2,1),mwrk(2,2),mwrk(2,3)

?print*,mwrk(3,1),mwrk(3,2),mwrk(3,3)

?]

END COLLECT
?list,group,prnt

*G #1 = PRNT
SET,GROUP,PRNT [
MESSAGE,"CONTENTS OF MWRK"
DISPLAY,#LINE
PRINT*,MWRK(1,1),MWRK(1,2),MWRK(1,3)
PRINT*,MWRK(2,1),MWRK(2,2),MWRK(2,3)
PRINT*,MWRK(3,1),MWRK(3,2),MWRK(3,3)
]
?set,breakpoint,1.7 [

IN COLLECT MODE
?read,prnt

?]

END COLLECT
?set,breakpoint,1.8 [

IN COLLECT MODE
?read,prnt

?]

END COLLECT
?set,breakpoint,1.9 [

IN COLLECT MODE
?read,prnt

?]

END COLLECT
?go
```

Establish group PRNT. The commands in this group display the values of array MWRK.

List the definition of group PRNT.

Set breakpoints at lines 7, 8, and 9. In each breakpoint body, include a READ command to initiate execution of the commands in PRNT.

Initiate program execution.

Figure 5-4. Program MATOP and Debug Session Illustrating Command Group Execution (Sheet 1 of 2)

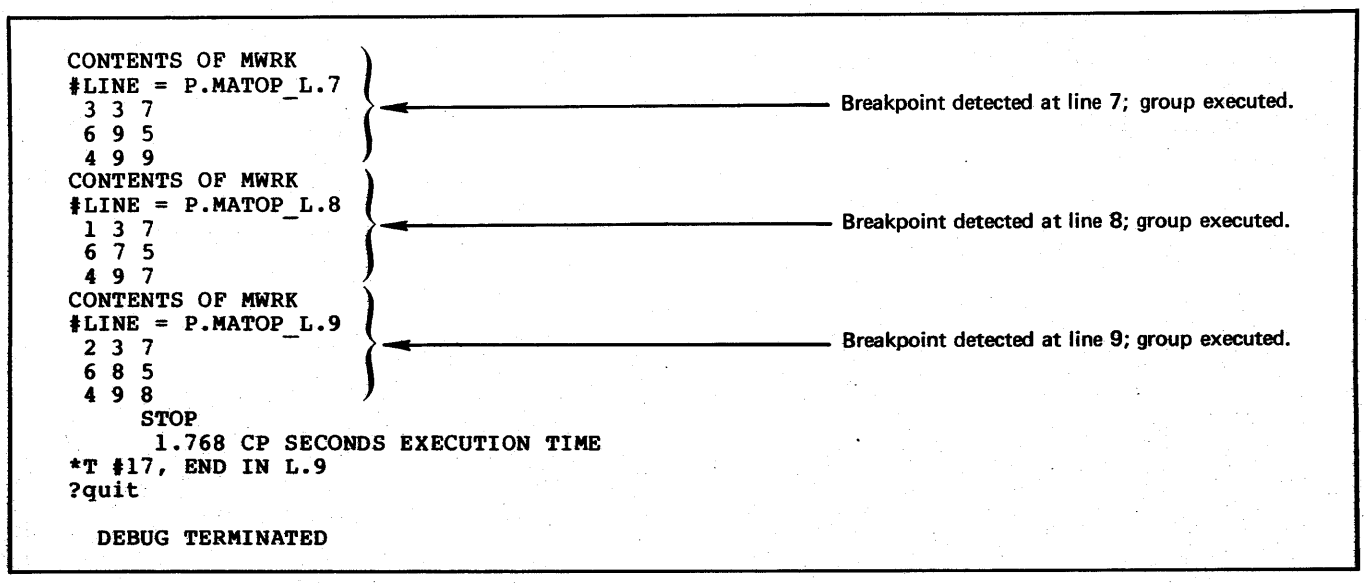

Figure 5-4. Program MATOP and Debug Session Illustrating Command Group Execution (Sheet 2 of 2)

An example of error processing during sequence execution is illustrated in figure 5-6. When the group named CGR is executed CID issues an error message and a warning message. In response to the error message, NO is entered instructing CID to ignore the command and resume execution of the sequence. In response to the warning message, a new command is entered; CID automatically executes the new command and resumes execution of the sequence.

RECEIVING CONTROL DURING SEQUENCE EXECUTION

Normally, a command sequence executes to completion without returning control to CID. There might be instances, however, when you would like to temporarily gain control during execution of a sequence for the purpose of entering other commands. You can do this either with the PAUSE command or by issuing a terminal interrupt (described later in this section).

PAUSE COMMAND

The PAUSE command can be included in a command sequence to suspend execution of the sequence at that point. The formats of the PAUSE command are:

```
PAUSE
PAUSE,"string"
```

where string is any string of characters. When CID encounters this command in a sequence, execution of the sequence is suspended and CID gets control, allowing you to enter commands. If string is specified, the character string is displayed.

The PAUSE command can be issued only from a command sequence; it cannot be entered directly from the terminal.

When a PAUSE command is issued from a trap or breakpoint body, CID displays the trap or breakpoint message, followed by any message included in the PAUSE command, and prompts for user input.

Execution of the suspended sequence can be resumed by either the GO or the EXECUTE command. These commands are explained in the following paragraphs.

GO AND EXECUTE COMMANDS

When execution of a command sequence has been suspended because of a PAUSE command, you can either resume execution of the sequence with the GO command or return control to your program with the EXECUTE command. The functions of these commands are summarized as follows:

- EXECUTE always resumes execution of the user program regardless of whether the command is issued following suspension because of a trap or breakpoint, following suspension because of a PAUSE command, or from a command sequence.
- GO resumes execution of the most recently suspended process. If issued following occurrence of a trap or breakpoint, GO resumes execution of the user program. If issued following a PAUSE command, GO resumes execution of the suspended sequence. If issued from a sequence, GO resumes execution of the sequence or program that had control when sequence execution was initiated.

When issued from the terminal following detection of a PAUSE command, GO resumes execution of the suspended command sequence. Following is an example of a PAUSE command issued from a sequence:

```
SET,BREAKPOINT,L.16 [
MESSAGE,"BREAKPOINT AT LINE 16"
X=0.0
PRINT*,A,X
PAUSE,"SEQUENCE EXECUTION SUSPENDED"
A=0.0
]
```

This definition establishes a breakpoint with a body at line 16 of the home program. When the breakpoint is encountered during program execution, execution of the

```
CYBER INTERACTIVE DEBUG
?set,group,prnt [

IN COLLECT MODE
?message,"contents of mwrk"

?display,#line

?print*,mwrk(1,1),mwrk(1,2),mwrk(1,3)

?print*,mwrk(2,1),mwrk(2,2),mwrk(2,3)

?print*,mwrk(3,1),mwrk(3,2),mwrk(3,3)

?]

END COLLECT
?set,breakpoint,l.7

?set,breakpoint,l.8

?set,breakpoint,l.9

?go

*B #1, AT L.7
?read,prnt

CONTENTS OF MWRK
#LINE = P.MATOP_L.7
 3 3 7
 6 9 5
 4 9 9
?go

*B #2, AT L.8
?read,prnt

CONTENTS OF MWRK
#LINE = P.MATOP_L.8
 1 3 7
 6 7 5
 4 9 7
?go

*B #3, AT L.9
?read,prnt

CONTENTS OF MWRK
#LINE = P.MATOP_L.9
 2 3 7
 6 8 5
 4 9 8
?go

*T #17, END IN L.9
?
     STOP
      1.810 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Establish a group to print the values of array MWRK.

Set breakpoints at lines 7, 8, and 9.

Initiate execution of group PRNT while program execution is suspended at line 7.

Initiate execution of group PRNT while program execution is suspended at line 8.

Initiate execution of group PRNT while program execution is suspended at line 9.

Figure 5-5. Debug Session Illustrating READ Command Entered at the Terminal

command sequence begins: an informative message is displayed, the variable X is set to 0.0, and the values of A and X are displayed; the PAUSE command suspends execution of the sequence, displays the specified string, and gives interactive control to the user. At this point, two processes are suspended: the user program and the command sequence. If a GO command is issued from the terminal, sequence execution resumes: the variable A is set to 0.0 and control automatically returns to the user program at line 16. If, however, an EXECUTE command is issued from the terminal, control immediately transfers to line 16 of the user program and execution resumes.

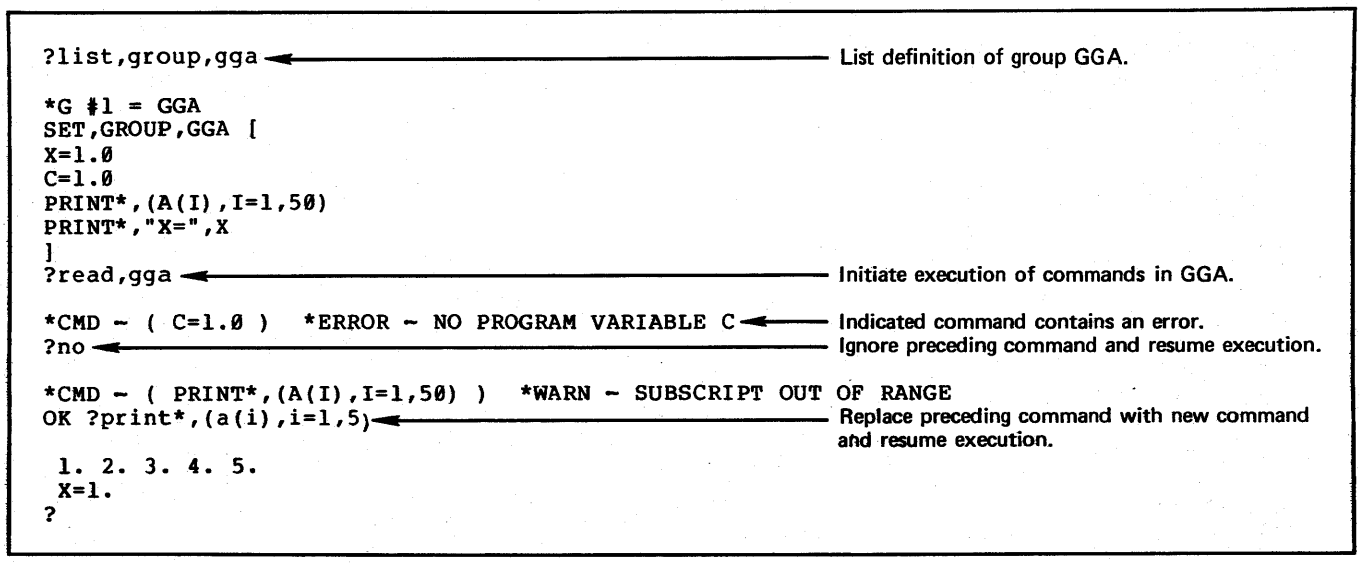

Figure 5-6. Debug Session Illustrating Error Processing During Sequence Execution

When issued from within a command group, GO causes an immediate exit from the sequence and a resumption of the process that was active when the sequence was invoked. If the READ command that called the group was issued from the terminal, a ? prompt is displayed and interactive mode is resumed. If the READ was issued from another command sequence (group, trap, or breakpoint body), execution of that sequence resumes at the command following the READ. For example, consider the following group definition:

```
SET,GROUP,TSTX [
X=X+DX
IF(X.LT.1000.0) GO
X=0.0
]
```

If the command READ,TSTX is issued from the terminal, then the commands in the sequence are executed: the current value of X is replaced by X+DX and is tested. If X is greater than or equal to 1000.0, sequence execution continues with the command X=0.0 and CID gets control. If X is less than 1000.0, the GO command immediately transfers control to CID. If the IF(X.LT.1000.0) GO command is replaced with IF(X.LT.1000.0) EXECUTE, then program execution immediately resumes (CID does not get control).

Now suppose the READ command is issued from another sequence, as in the following example:

```
SET,BREAKPOINT,L.5 [
READ,TSTX
LIST,VALUES
]
```

The READ command is executed automatically when the breakpoint at line 5 is detected, which initiates execution of the commands in TSTX. If X is less than 1000.0, the GO command resumes execution of the most recently suspended process, in this case the commands in the breakpoint body: LIST,VALUES is executed and program execution resumes at line 5. If the IF(X.LT.1000.0) GO command is replaced with IF(X.LT.1000.0) EXECUTE, execution of the user program immediately resumes at line 5 in this case.

The debug session in figure 5-7 illustrates the PAUSE command. This session was produced by executing program AREA, shown in figure 3-1, under CID control. The purpose of this session is to suspend execution at the beginning of the subroutine in order to display the input values and change them if necessary and to suspend execution at the end of the subroutine in order to display the calculated area. To accomplish this, a breakpoint with a body is set at line 2 of subroutine AREA. Two commands are included in the body: a PRINT command and a PAUSE command. A STORE trap is then established for the variable A. A body containing a command to print the value of A is defined for this trap. On each of the three passes through subroutine AREA, the commands in the sequence are executed automatically. The PAUSE command suspends execution of the breakpoint body. When the PAUSE command is detected on the first pass, GO is entered to resume sequence execution. (In this case, GO and EXECUTE have the same effect since PAUSE is the last command in the sequence.) On the next two passes through the subroutine, assignment commands are entered to change the values of some of the input variables while execution is suspended because of the PAUSE command.

Both the GO and EXECUTE commands can be used to resume program execution at a location other than the one where execution was suspended. The command forms are:

```
GO,loc
EXECUTE,loc
```

where loc is a program address of the form L.n or S.n. These command forms resume execution at the location designated by loc.

These commands can be used to skip sections of code, as illustrated in figure 5-8. In this example, the main program passes two values A and B to a subroutine which calculates a value for C. C is then used in a subsequent calculation. The programmer wishes to skip the call to SUB, assigning instead his own value to C. A breakpoint is set at line 4 to suspend execution immediately before execution of the CALL statement. When execution is suspended at the breakpoint location, a value is assigned to C. Execution is then resumed at line 5, and line 4 is bypassed.

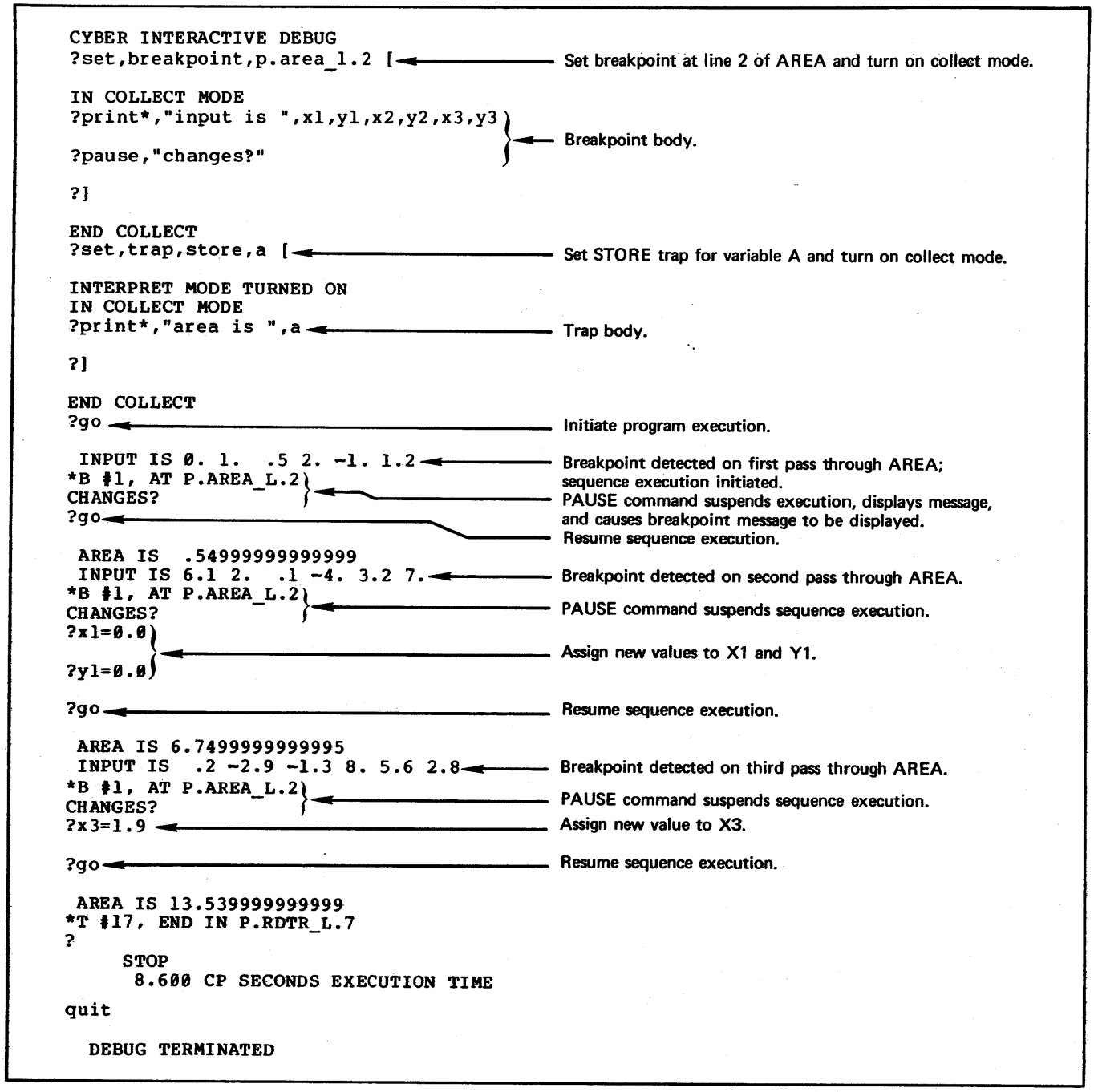

Figure 5-7. Debug Session Illustrating PAUSE Command

CONDITIONAL EXECUTION OF CID COMMANDS

CID allows conditional execution of commands in much the same manner as FORTRAN Extended does for executable statements. CID provides an IF command that is similar to the FORTRAN IF statement and a JUMP command that is similar to the FORTRAN GO TO statement.

IF COMMAND

The format of the IF command is:

IF (expr) command

where expr is a relational expression and command is any valid CID command. If the relational expression is true, CID executes the command.

```
1            PROGRAM EX         74/74   TS ID

                          PROGRAM EX(OUTPUT)
                          A=1.0
                          B=2.0
                          CALL SUB(A,B,C)
         5                D=C**2+1.0
                          WRITE 100,A,B,C,D
                  100     FORMAT(" VALUES ARE ",4F6.2)
                          STOP
                          END

                          SUBROUTINE SUB(A,B,C)
                          C=A+B
                          RETURN
                          END

CYBER INTERACTIVE DEBUG
?set,breakpoint,l.4

?go

*B #1, AT L.4 <-------------------------- Execution suspended at line 4 of main program.
?c=4.0 <-------------------------- Assign value to C.

?go,l.5 <-------------------------- Resume execution at line 5.

VALUES ARE   1.00  2.00  4.00 17.00
*T #17, END IN L.8
?
     STOP
       .149 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Figure 5-8. Program EX and Debug Session Illustrating GO Command

The form of a relational expression is the same as in FORTRAN. The following relational operators are valid:

.EQ.	.GT.
.NE.	.LE.
.LT.	.GE.

The following restrictions apply to the IF command:

Only variables defined in the current home program can appear.

CID variables cannot be used.

Function references and exponentiation are not allowed.

Address qualification is not allowed.

Although the consequent command in an IF can be any valid CID command, it is usually an assignment, PRINT, JUMP, or GO command, as in the following examples:

IF(X.GT.Y+Z)PRINT*,"VALUES ARE",X,Y
Print the values of X and Y if X is greater than Y+Z.

IF(IFIRST.EQ.1)ZZ=XX*2.0
IF IFIRST is equal to 1, then the current value of ZZ is replaced by the value XX times 2.0.

IF(A(I).GT.0.0)GO,L.50
If the value of A(I) is greater than zero, control transfers to line 50 of the program.

IF(A(2).NE.B(2))JUMP,LABL
If the value of A(2) does not equal the value of B(2), control transfers to the commands following the label LABL in the current command sequence.

Although you can issue an IF command from the terminal, as described in section 3, this command is especially powerful when used in command sequences. You can use the IF command to perform a test and to conditionally transfer control to another statement in the sequence or exit from the sequence. The technique for doing this is similar to that of FORTRAN. In FORTRAN, a GO TO statement causes a branch to another executable statement. In CID, the GO or EXECUTE command is used to exit from the current sequence, as in the following examples:

IF(A.GT.B)GO
If the value of A is greater than the value of B, exit from the current sequence and resume execution of the most recently suspended process.

IF(I.NE.0)GO,S.20
If the value of I is not equal to zero, exit from the current sequence and resume program execution at statement 20.

IF(X+T.LT.Y+S)EXECUTE
If the value of X+T is less than the value of Y+S, exit from the current sequence and resume program execution.

The JUMP and LABEL commands are used to transfer control within a sequence.

JUMP AND LABEL COMMANDS

The format of the JUMP command is:

```
JUMP,name
```

where name is a label declared in a LABEL command. The function of the JUMP command is identical to the FORTRAN GO TO statement. When CID encounters a JUMP command, control transfers to the command following the label.

A label is established within a command sequence by the following command:

```
LABEL,name
```

where name is a string of one through seven letters or digits. The LABEL command is not executed by CID; its sole purpose is to provide a destination for a JUMP command. When a JUMP command is executed, control transfers to the command following the LABEL command.

The JUMP command can be used in conjunction with the IF command to perform a conditional branch, as in the following command sequence example:

```
IF(X.LT.100.0)JUMP,LAB1
X=0.0
GO
LABEL,LAB1
X=X+1.0
```

If the value of X is less than 100.0, 1.0 is added to X and program execution resumes; if X is not less than 100.0, X is set to 0.0 and program execution resumes.

A debug session using the IF, JUMP, and LABEL commands is illustrated in figure 5-9. The program executed to produce this session appears in figure 3-4. The purpose of this session is to suspend program execution at the beginning of subroutine SETB and store the value 3.0 into each word of the array B if K is equal to 3. If K is not equal to 3, execution is to proceed normally. To accomplish this, a breakpoint with a body is set at line 3 of SETB. The first command in the body tests K: if K is not equal to 3, program execution resumes at line 3; otherwise, execution of the sequence continues. The remaining commands of the sequence constitute a loop that stores 3.0 into B. The variable K is used as an index and counter since it is not required by the program. When K is equal to the array dimension N, program execution resumes at line 9. A breakpoint is set in the main program at the first subroutine call so that K can be assigned a value of 3.

As you are undoubtedly aware by now, command sequences using the conditional execution capability can become quite complicated. You should, however, attempt to keep sequences short and simple so that you don't spend more time debugging the sequence than would be required to debug your program.

COMMAND FILES

In addition to executing command sequences established within a debug session, you can execute command sequences stored on a separate file. You can create such a file using a text editor and include any sequence of CID commands in the file. Command files can also be created with the SAVE command (discussed under Saving Trap, Breakpoint, and Group Definitions). There are two reasons why you might want to create a separate file of CID commands:

- By storing commands on a file you have a permanent copy of the command sequence that can be used for future debug sessions.
- Editing a file of commands using a text editor is much easier than editing a sequence of commands in a group or body while executing under CID control. (See Editing a Command Sequence.)

To execute the commands in a file, enter the command:

```
READ,lfn
```

where lfn is the file name. CID reads the file and automatically executes the commands, just as for a body or group. Control returns to CID when execution of the sequence is complete.

Executing commands from a file can be time-consuming if the sequence is executed many times since the file must be read each time the sequence is executed. If a command sequence is to be executed many times in a single session, a more efficient method of executing the commands is to create a command file containing a SET,GROUP command and to include the command sequence in the group. When the file is read by the READ command, the SET,GROUP command is automatically executed and the command sequence is established as a group within the debug session. The group can subsequently be executed without the necessity of reading the file. For example, suppose a file containing the commands:

```
X1=Y1+Z1
X2=Y2+Z2
PRINT*,X1,X2
```

is created via a text editor and assigned the name COMF. The command READ,COMF must be issued whenever the sequence is to be executed. Now suppose instead the following file is created:

```
SET,GROUP,GRPX [
X1=Y1+Z1
X2=Y2+Z2
PRINT*,X1,X2
]
```

Then the command READ,COMF reads the file and causes the SET,GROUP command to be executed, establishing GRPX for the current session. Thereafter, the command READ,GRPX executes the commands in the group and the file COMF is only read once.

The use of NOS and NOS/BE text editors to create and edit files containing CID commands is discussed under Editing a Command Sequence.

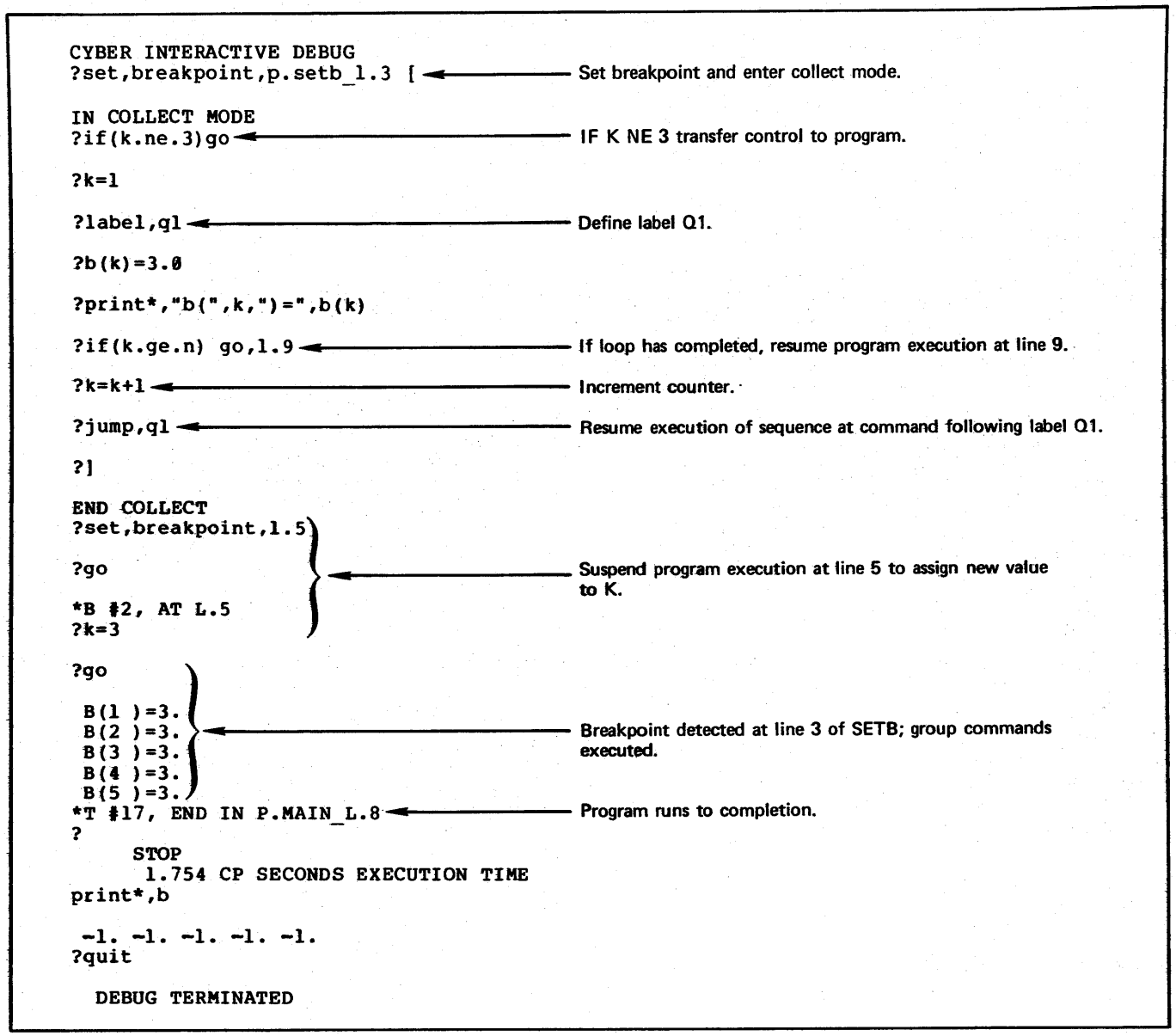

Figure 5-9. Debug Session Illustrating JUMP and LABEL Commands

SAVING TRAP, BREAKPOINT, AND GROUP DEFINITIONS

As with other CID commands, command sequences exist only for the duration of the session in which they are defined. CID provides the capability of saving group, trap, and breakpoint definitions on a separate file. You can print this file or make it permanent. There are two basic reasons for copying CID definitions to a file:

- To preserve a copy of the definitions for use in the current or in subsequent debug sessions.
- To facilitate the editing of a command sequence with the system text editor.

The command to save CID definitions has the following forms:

SAVE,BREAKPOINT,lfn,list
Copy to file lfn the definitions of the breakpoints specified in list; list is an optional list of breakpoint locations (S.n or L.n) or breakpoint numbers (#n) separated by commas. If list is omitted all breakpoints are saved.

SAVE,TRAP,lfn,type,scope
Copy to file lfn the definitions of the traps of the specified type defined for the specified scope. Type and scope are optional and are the same as for the SET,TRAP command.

SAVE,GROUP,lfn,list
Copy to file lfn the groups specified in list; list is an optional list of group names or numbers (#n) separated by commas. If list is omitted, all groups defined for the current session are saved.

The SAVE command copies the complete definition of the specified traps, breakpoints, or groups, including the original SET command, to the specified file. When a SAVE command is executed, the file is written, closed, and rewound. Therefore, you should specify a unique file name for each SAVE command issued during a debug session; otherwise, the original information is overwritten. You can, however, save an entire CID environment, including all trap, breakpoint, and group definitions, on a single file by issuing the command:

SAVE,,lfn

This is the only form of the SAVE command that allows you to mix trap, breakpoint, and group definitions on a single file.

Some examples of SAVE commands are as follows:

SAVE,BREAKPOINT,SBPF
Copy to file SBPF all breakpoints currently defined.

SAVE,BREAKPOINT,BPFILE,L.10,P.SUBX_S.20
Copy to BPFILE the definitions of the breakpoints established at line 10 of the home program and statement 20 of subroutine SUBX.

SAVE,BREAKPOINT,FILEA,#2,#5
Copy to FILEA the definition of breakpoints #2 and #5.

SAVE,TRAP,TFILE
Copy to TFILE all traps currently defined.

SAVE,TRAP,TTT,RJ,P.PROGA
Copy to TTT the definition of the RJ trap established in program unit PROGA.

SAVE,GROUP,GFIL,WRT,RDD,GRPX
Copy to GFIL the definitions of the groups named WRT, RDD, and GRPX.

Definitions stored on a file can be altered (as described under Editing a Command Sequence) and then restored in the current or in a subsequent session. The command to restore the definitions stored on a file is:

READ,lfn

where lfn is the file containing the definitions. You can issue a READ command in the current session or in a later session. If a READ,lfn is issued in the current session, and the definitions previously saved on lfn have not been removed by the appropriate CLEAR command, CID displays a message of the form:

EXISTING BREAKPOINTS WILL BE REDEFINED
OK?

A positive response (YES or OK) causes the existing definitions to be redefined according to the information in the file; a negative response (NO) causes the read command to be ignored.

The following READ commands assume that GFIL and TTT are as defined in the preceding example:

READ,TTT
Restore the RJ trap definition contained in file TTT.

READ,GFIL
Restore the group definitions contained in file GFIL.

A debug session using the SAVE command is illustrated in figure 5-10. The program shown in figure 3-1 is executed to produce this session. A breakpoint with a body is established in the main program and in subroutine AREA, after which execution is initiated. The program reads the three records contained in TRFILE. On each pass through the program, the command sequences are executed. After the program terminates, CID gets control because of the END trap, and a SAVE,BREAKPOINT command is issued to save the current breakpoint definitions on the file named AFILE. The session is terminated, the binary file LGO is rewound, and a new session is initiated. The command READ,AFILE restores the breakpoints for the new session. The contents of AFILE are shown in figure 5-11.

The debug sessions in figure 5-12 illustrate the SAVE,GROUP command using the program shown in figure 5-4. The command group PRNT, shown in figure 5-4, is saved on the file named GFILE at the end of the first debug session. At the beginning of the second session, the command READ,GFILE restores the group definition. Breakpoints are set at lines 7, 8, and 9 of MATOP. When each breakpoint is encountered the command READ,PRNT is issued to execute the group. Note that this command could have been placed in a body for each breakpoint. The groups would then have been executed automatically, without intervention from the user.

EDITING A COMMAND SEQUENCE

When you detect an error in or wish to make a change to a command sequence in a trap body, breakpoint body, or group you can remove the definition with the appropriate CLEAR command and reenter the entire sequence. This procedure can be time-consuming for lengthy sequences, however.

CID provides two alternate methods of making changes to a command sequence:

- You can save the trap, breakpoint, or group definition on a separate file and edit the file.
- You can turn on veto mode (described later in this section) and edit the sequence interactively.

To apply the first method you must temporarily exit from the current debug session.

First Session: Breakpoint Definitions Saved.

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.4 [

IN COLLECT MODE
?display,#line

?list,values,p.rdtr

?]

END COLLECT
?set,breakpoint,l.6

?set,breakpoint,p.area_l.7 [

IN COLLECT MODE
?display,#line

?print*,"area is",a

?]

END COLLECT
?go

#LINE = P.RDTR_L.4
P.RDTR
A = -I,   X1 = 0.0,   X2 = .5,   X3 = -1.0,   Y1 = 1.0,   Y2 = 2.0
Y3 = 1.2
#LINE = P.AREA_L.7
 AREA IS .54999999999999
*B #2, AT L.6  (OF P.RDTR)
?save,breakpoint,afile

?quit

  DEBUG TERMINATED
```

Set breakpoint with body at line 4 of main program.

Set breakpoint at line 6 of main program.

Set breakpoint with body at line 7 of subroutine AREA.

Initiate program execution.

Breakpoint detected at line 4; sequence execution initiated.

Breakpoint detected at line 7 of AREA; sequence execution initiated.

Breakpoint detected at line 6.

Copy breakpoint definitions to file AFILE.

Second Session: Breakpoint Definitions Restored.

```
..rewind,lgo

..lgo

CYBER INTERACTIVE DEBUG
?read,afile

?list,breakpoint

*B #1 = L.4 ,   *B #2 = L.6,   *B #3 = P.AREA_L.7
?

?go

#LINE = P.RDTR L.4
P.RDTR
A = -I,   X1 = 0.0,   X2 = .5,   X3 = -1.0,   Y1 = 1.0,   Y2 = 2.0
Y3 = 1.2
#LINE = P.AREA_L.7
 AREA IS .54999999999999
*B #2, AT L.6  (OF P.RDTR)
?quit

  DEBUG TERMINATED
```

Restore breakpoint definitions contained in AFILE.

List breakpoint locations.

Initiate program execution.

Figure 5-10. Debug Sessions Illustrating SAVE Command

```
SET HOME P.RDTR
SET,BREAKPOINT,L.4 [
DISPLAY,#LINE
LIST,VALUES,P.RDTR
]
SET HOME P.RDTR
SET BREAKPOINT L.6
SET,BREAKPOINT,P.AREA_L.7 [
DISPLAY,#LINE
PRINT*,"AREA IS",A
]
```

Figure 5-11. Listing of Breakpoint File AFILE

SUSPENDING A DEBUG SESSION

CID provides the capability of suspending the current session, returning to system command mode, and then resuming the session at a later time. This feature can be used whenever you wish to perform a function outside of CID, but it is especially useful for leaving a session to edit a command sequence.

The command:

SUSPEND

suspends the current session; copies the complete definition of the session environment, including trap, breakpoint, and group definitions to a file; and returns control to the operating system.

To resume the suspended session, issue the command:

DEBUG(RESUME)

This command restores the session to its status as it existed at the time of suspension. All traps, breakpoints, and groups have their original definitions, and all program and debug variables have their original values.

You do not have access to the file created by a SUSPEND and you cannot use SUSPEND and RESUME to continue a debug session at a subsequent terminal session. Once you have logged out, values of program and debug variables are lost. Group, trap, and breakpoint definitions can be recovered only if you have used the SAVE command to write them to a file and have made the file permanent.

DEBUG(RESUME) does not restore the status of any files attached to your program. You should therefore avoid any operations, such as REWIND, that would change the status of these files if you intend to continue the debug session. You should not attempt to execute or modify the program you are debugging while the session is suspended.

EDITING TRAP BODIES, BREAKPOINT BODIES, AND GROUPS

To edit a trap body, breakpoint body, or command group, proceed as follows:

1. Save the trap, breakpoint, or group definition with the appropriate SAVE command.
2. Suspend the current session with the SUSPEND command.
3. Use an editor to alter the command sequence.
4. Resume the session with the DEBUG(RESUME) command.
5. Remove the current trap, breakpoint, or group definition with the appropriate CLEAR command.
6. Establish the altered definition with the READ command.

Be sure that you do not alter the status of any files attached to your program while the session is suspended, as the DEBUG(RESUME) command does not restore these to their status at suspension time.

An example of the procedure for editing a command sequence is shown as performed under NOS/BE (figure 5-13) and NOS (figure 5-14). The purpose of this editing session is to change the command IF(I.EQ.0)X=Y contained in the group named AGRP to IF (I.EQ.1) X=Y.

To accomplish this, the debug session is suspended and the group is saved on the file named GRPFIL.

Under NOS/BE INTERCOM, the command EDITOR calls the system editor. GRPFIL is made the edit file by the command EDIT,GRPFIL,SEQ. This command also assigns a sequence number to each line in the editing file. The command 140=IF(I.EQ.1) replaces the current contents of line 140. The current GRPFIL is destroyed, a new one is created from the current edit file, and edit mode is terminated. Refer to the NOS/BE Interactive Guide for Users of FORTRAN Extended for detailed information on the INTERCOM text editor.

Under NOS, the following commands are used to alter GRPFIL:

EDIT,GRPFIL
: Enter edit mode to edit GRPFIL.

LIST;7
: List 7 lines.

SET;4
: Advance the line pointer 4 lines.

CHANGE
: Replace the line indicated by the line pointer by the string enclosed in slashes.

LIST
: List the line indicated by the line pointer.

END
: Exit from edit mode.

When editing is complete, the debug session is resumed by DEBUG(RESUME) and the group is restored by READ,GRPFIL. Refer to the NOS Text Editor reference manual for detailed information on the NOS text editor.

First Session: Group Defined and Saved.

```
CYBER INTERACTIVE DEBUG
?set,group,prnt [
IN COLLECT MODE
?message,"contents of mwrk"
?display,#line
?print*,mwrk(1,1),mwrk(1,2),mwrk(1,3)
?print*,mwrk(2,1),mwrk(2,2),mwrk(2,3)
?print*,mwrk(3,1),mwrk(3,2),mwrk(3,3)
?]
END COLLECT
?save,group,gfile
?quit
  DEBUG TERMINATED
```

Assign group name and activate collect mode.

Group body; contains commands to display message, current line number, and contents of MWRK.

Copy group definition to file GFILE.

Second Session: Group Reestablished.

```
..rewind,lgo
..lgo
CYBER INTERACTIVE DEBUG
?read,gfile
?list,group
*G #1 = PRNT
?list,group,#1
*G #1 = PRNT
SET,GROUP,PRNT [
MESSAGE,"CONTENTS OF MWRK"
DISPLAY,#LINE
PRINT*,MWRK(1,1),MWRK(1,2),MWRK(1,3)
PRINT*,MWRK(2,1),MWRK(2,2),MWRK(2,3)
PRINT*,MWRK(3,1),MWRK(3,2),MWRK(3,3)
]
?set,breakpoint,1.7
?set,breakpoint,1.8
?set,breakpoint,1.9
?go
*B #1, AT L.7
?read,prnt
CONTENTS OF MWRK
#LINE = P.MATOP_L.7
 3 3 7
 6 9 5
 4 9 9
?
```

Restore group definitions contained in GFILE.

List current group names and numbers.

List commands in group #1.

Initiate execution of the commands in PRNT.

Figure 5-12. Debug Session Illustrating READ and SAVE,GROUP Commands

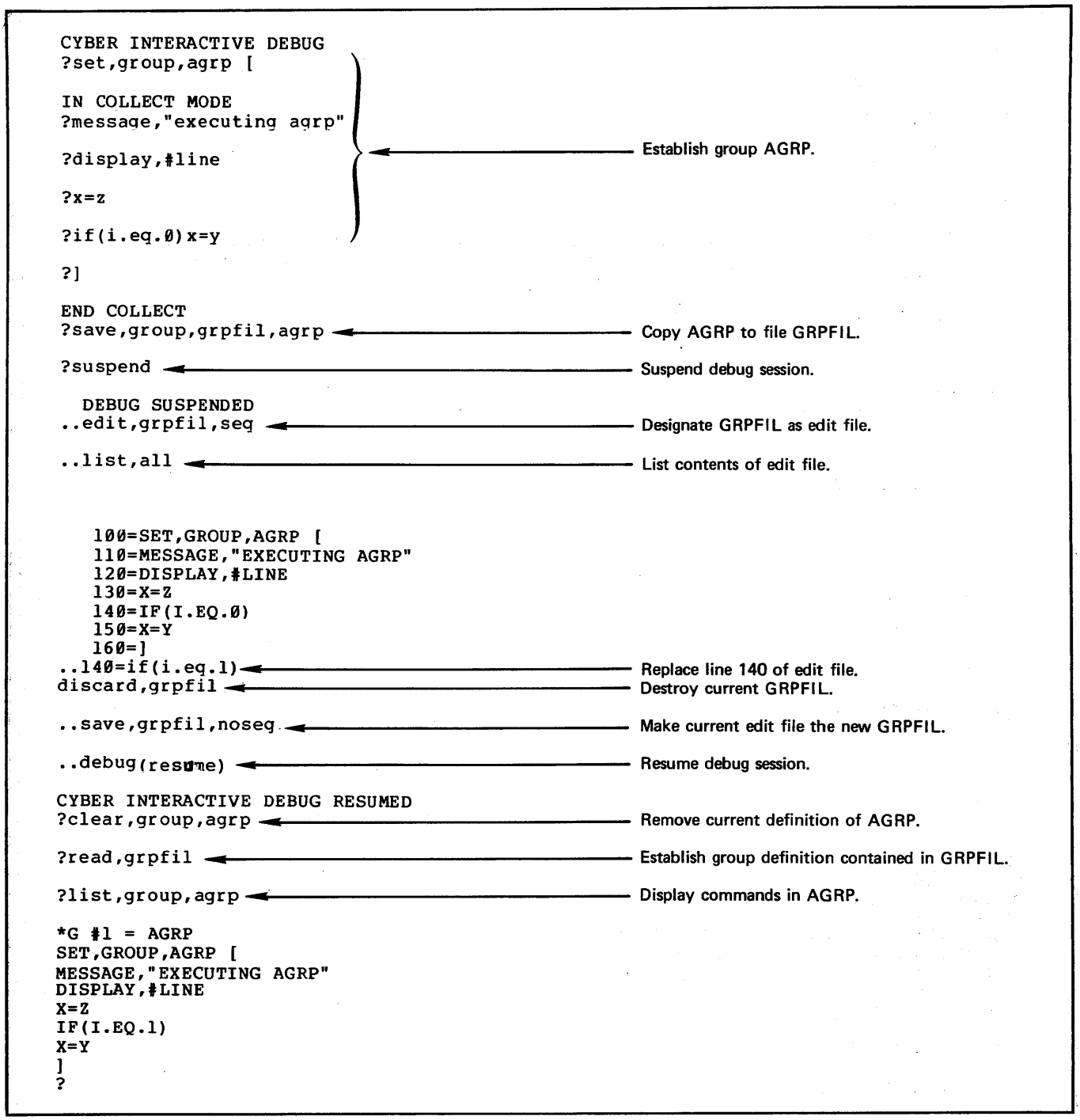

Figure 5-13. Editing a Command Sequence Using EDITOR Under NOS/BE INTERCOM

VETO MODE

Veto mode provides a method of interactively altering a command sequence. When veto mode is on, CID automatically displays each command in a sequence before it is executed and allows you to specify whether the command is to be executed as is, ignored, or replaced by another command.

To activate veto mode enter the command:

```
SET,VETO,ON
```

Once veto mode has been turned on, it remains on until turned off by one of the commands:

```
SET,VETO,OFF
```

or

```
CLEAR,VETO
```

Although veto mode is intended to be used with command sequences, commands entered interactively are also subject to veto.

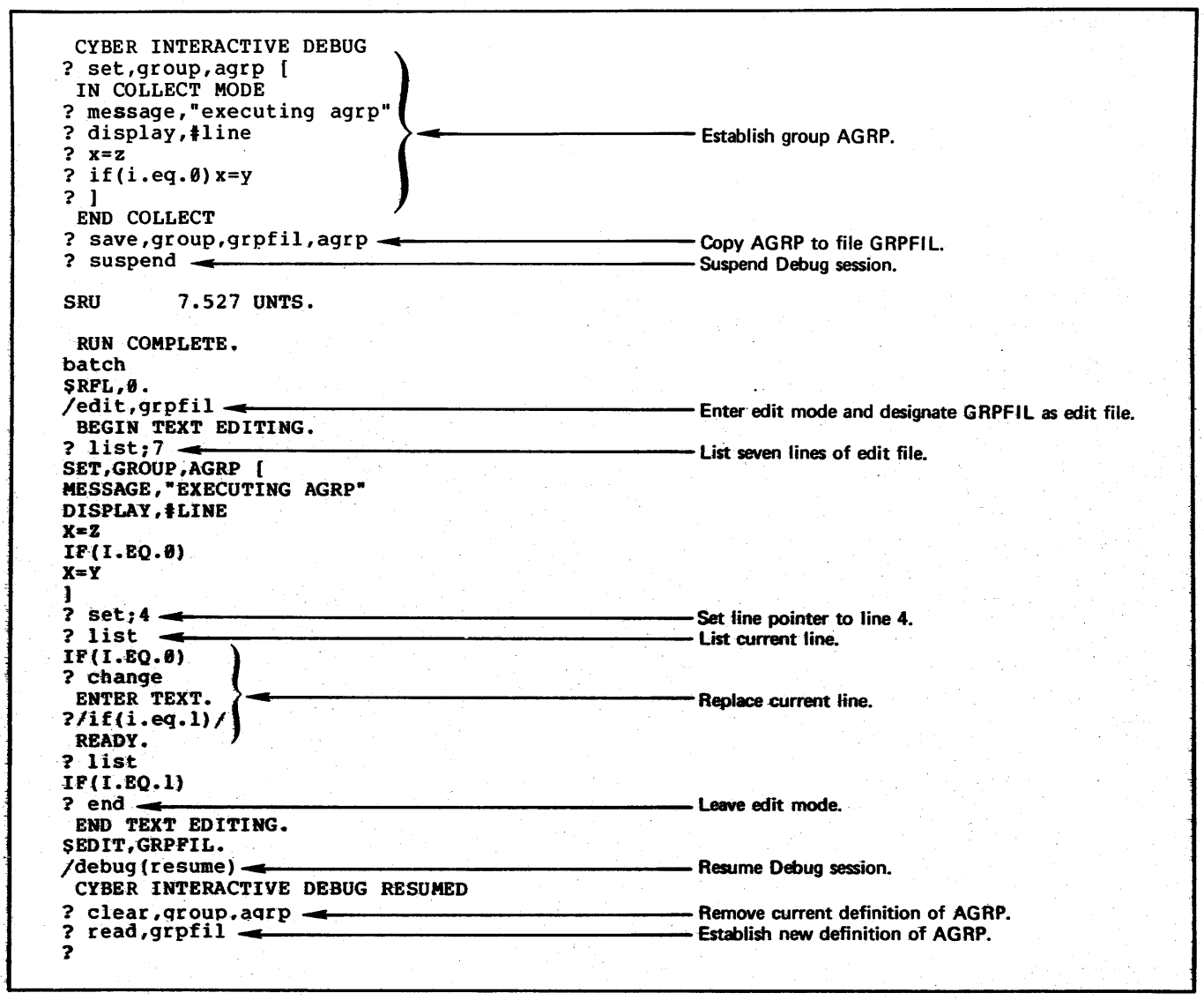

Figure 5-14. Editing a Command Sequence Using the EDIT Program Under NOS

When veto mode is on and CID encounters a sequence, CID displays each command just before it is executed and follows the command by the user prompt OK?. Valid responses and subsequent CID action are as follows:

User Response	Debug Action
YES or OK	Execute the command.
NO	Ignore the command.
YES,SEQ or OK,SEQ	Execute the command and inhibit veto mode for the remainder of the sequence.
NO,SEQ	Ignore the command and inhibit veto mode for the remainder of the sequence; veto mode resumes after the current sequence is completed.
Any CID command	Execute the specified command or commands in place of the current command. Note that the new command does not actually replace the current command in the sequence; if the sequence is executed again, the new command must be reentered.

An example of veto mode execution is illustrated in figure 5-15. The group named GRPZ contains the command X=X-1.0 which is replaced by the command X=X+1.0 immediately before it is executed. Note that all commands entered while veto mode is on can be vetoed. Note also that the commands JUMP,YPI and Y=Y+1.0 are not reached in the flow of execution and are therefore not displayed.

```
?list,group,grpz <-------- List the complete definition of group GRPZ.
*G #1 = GRPZ
SET,GROUP,GRPZ [
IF(X.GT.Y)
JUMP,YP1
X=X-1.0 <-------- Command to be replaced while in veto mode.
GO
LABEL,YP1
Y=Y+1.0
]
? print*,x,y
  1. 2.
? set,veto,on <-------- Activate veto mode.
? read,grpz <-------- Initiate execution of commands in GRPZ.
 OK ? ok <-------- Execute preceding command.
 *CMD - ( IF(X.GT.Y) ) <-------- First command in GRPZ.
 OK ? ok <-------- Execute preceding command.
 *CMD - ( X=X-1.0 ) <-------- Second command in GRPZ.
 OK ? x=x+1.0 <-------- Replace preceding command with this command.
 *CMD - ( GO ) <-------- Third command in GRPZ.
 OK ? ok <-------- Execute preceding command.
? set,veto,off <-------- Turn off veto mode.
 OK ? ok <-------- Execute preceding command.
? print*,x,y
  2. 2.
?
```

Figure 5-15. Debug Session Illustrating Veto Mode

DISPLAYING COMMAND SEQUENCES AS THEY EXECUTE

Normally, CID does not display the commands in a trap or breakpoint body or group as they are executed. In some cases, however, you might want to examine the commands in the sequence as they are executed. You can cause the commands in a sequence to be displayed whenever the sequence is invoked by issuing a SET,OUTPUT command and specifying the R or B options. The B option causes CID to display each command in a trap or breakpoint body immediately before the command is executed; the R option causes CID to display each command in a group or file immediately before the command is executed. You should remember to specify the default output types on the SET,OUTPUT command so that they are not suppressed. For example, the command:

```
SET,OUTPUT,E,W,I,D,R,B
```

displays each command of a command sequence whenever the sequence is executed, as well as the normal output (See section 4, Control of CID Output).

INTERRUPTS DURING SEQUENCE EXECUTION

You can obtain control at any time during a debug session by issuing a terminal interrupt. This is accomplished by keying a %A under NOS/BE, a BREAK under NOS, or a) under NOS IAF.

If your program is executing when you issue a terminal interrupt, an interrupt trap occurs as described in section 4. If a command sequence is executing at the time of the interrupt, execution of the sequence is suspended and CID issues the message:

```
INTERRUPTED
?
```

You can respond as follows:

User Response	Debug Action
OK or YES	Resume execution at the point of the interrupt.
GO or NO,SEQ	Disregard all remaining commands in the sequence and resume execution of the user program.
Any CID command	Execute the specified command and resume execution of the sequence at the point of the interrupt.

If CID is in the process of displaying information when the interrupt is issued, the information remaining to be printed is lost. A terminal interrupt is therefore an effective means of stopping excessive CID output.

EXAMPLES OF DEBUG SESSIONS USING COMMAND SEQUENCES

Following are two examples of debug sessions that use command sequences. The programs CORR and NEWT, debugged in section 3, are used to illustrate how sequences can be used to speed up the debugging process.

PROGRAM CORR

The original version of CORR, with errors, is shown in figure 3-16. Several debug sessions were required to debug the program completely. Commands issued during one session had to be reentered in subsequent sessions.

This example demonstrates how this repetition can be eliminated by including the assignment commands in trap and breakpoint bodies and saving the trap and breakpoint definitions on a separate file for use in later sessions. The example also demonstrates how an appropriate command sequence can be used to simulate the reading of input data.

The NOS/BE text editor is used to create the three command files shown in figure 5-16. Each file corresponds to a test case. The files contain assignment commands that insert the correct values for SUMYSQ and N and test values in the arrays X and Y. Two commands, separated by a semicolon, are included on each line. The files are named TEST1, TEST2, and TEST3, respectively.

Listing of TEST1:

```
SUMYSQ=0.0;N=5
X(1)=10.0;Y(1)=10.1
X(2)=20.5;Y(2)=21.1
X(3)=6.0;Y(3)=6.1
X(4)=34.0;Y(4)=32.9
X(5)=4.4;Y(5)=4.5
```

Listing of TEST2:

```
SUMYSQ=0.0;N=5
X(1)=1.0;Y(1)=2.0
X(2)=5.0;Y(2)=5.0
X(3)=10.0;Y(3)=10.0
X(4)=2.0;Y(4)=2.0
X(5)=7.0;Y(5)=7.0
```

Listing of TEST3:

```
SUMYSQ=0.0;N=4
X(1)=2.0;Y(1)=1.0
X(2)=2.0;Y(2)=5.1
X(3)=2.0;Y(3)=7.6
X(4)=2.0;Y(4)=10.0
X(5)=0.0;Y(5)=0.0
```

Figure 5-16. Command Files Initializing Input Variables for Program CORR

Another file named BPFILE, shown in figure 5-17, is created using the text editor. This file contains two breakpoint definitions. The first breakpoint is set at line 14. The PAUSE command will temporarily suspend execution of the breakpoint body. The user will then issue a READ command to execute the commands in TEST1. The command GO,L.22 will resume program execution at line 22, skipping the FORTRAN READ statement. The second breakpoint is set at line 32. The body of this breakpoint contains assignment commands that will calculate and insert the correct values for SUMXY and RSQ when the body is executed.

The debug sessions for program CORR are shown in figure 5-18. One session is conducted for each test case. At the beginning of each session the command READ,BPFILE is issued to establish the breakpoint definitions and program execution is initiated. When the PAUSE command gives interactive control to the user, a READ command is issued to load the arrays X and Y. Execution is resumed by the GO command, and the commands in the sequences are executed automatically. A LIST,BREAKPOINT is issued in the first session to display existing breakpoints and bodies.

PROGRAM NEWT

The command sequence capability can be applied to the debugging of the Newton's method subroutine shown in figure 3-25. Two of the errors in the original program involved an incorrect function name in the subroutine call and an incorrect convergence check that resulted in an infinite loop. The CID commands to correct these errors can be placed in a breakpoint body, as shown in figure 5-19. The sequence includes commands to calculate the correct functional value FX, print the functional value and current number of iterations, test for convergence, and resume program execution at a location following the erroneous statements. If the convergence criterion is satisfied, the PAUSE command will suspend execution of the sequence. The breakpoint, set at line 400, is encountered on each pass through the loop. Note that although line 400 is illegal because of the unresolved function reference, program execution will be suspended before the statement is executed. The subsequent GO command will resume execution at line 420, bypassing the illegal statement.

After the breakpoint is defined, program execution is initiated with the GO command. The commands in the breakpoint body are executed automatically, as indicated by the PRINT command output, until the convergence criterion is satisfied. In response to the first occurrence of the PAUSE command, GO is entered to resume program execution. In response to the next occurrence of PAUSE, the command EXECUTE(P.MAIN,L.150) is entered to transfer control to line 150 of the main program, allowing the program to print the results and to terminate.

```
SET,BREAKPOINT,L.14 [
PAUSE,"INPUT?"
PRINT*,"X=",X," Y=",Y
GO,L.22
]
SET,BREAKPOINT,L.33 [
SUMXY=X(1)*Y(1)+X(2)*Y(2)
SUMXY=SUMXY+X(3)*Y(3)+X(4)*Y(4)
SUMXY=SUMXY+X(5)*Y(5)
SUMX=(N*SUMXY-SUMX*SUMY)*(N*SUMXY-SUMX*SUMY)
IF(DENOM.EQ.0)PAUSE,"DENOM IS 0"
RSQ=SUMX/DENOM
]
```

Set breakpoint with body at line 14; when PAUSE is executed, user can issue command to read command file.

Set breakpoint with body at line 32; commands are included to calculate correct values for SUMXY and RSQ and to test DENOM for zero value.

Figure 5-17. Listing of File BPFILE Containing Breakpoint Definitions

Debug Session for First Test Case:

```
CYBER INTERACTIVE DEBUG
?read,bpfile                                  <- Establish breakpoint definitions stored in BPFILE.

?list,breakpoint                              <- List breakpoint locations.

*B #1 = L.14 ,   *B #2 = L.33
?list,breakpoint,#1,#2                        <- List breakpoint bodies.

*B #1 = L.14
SET,BREAKPOINT,L.14 [
PAUSE,"INPUT?"                                <- Breakpoint #1.
PRINT*,"X=",X," Y=",Y
GO,L.22
]
*B #2 = L.33
SET,BREAKPOINT,L.33 [
SUMXY=X(1)*Y(1)+X(2)*Y(2)
SUMXY=SUMXY+X(3)*Y(3)+X(4)*Y(4)
SUMXY=SUMXY+X(5)*Y(5)
SUMX=(N*SUMXY-SUMX*SUMY)*(N*SUMXY-SUMX*SUMY)  <- Breakpoint #2.
IF(DENOM.EQ.0)
PAUSE,"DENOM IS 0"
RSQ=SUMX/DENOM
]
?go

*B #1, AT L.14                                <- Breakpoint detected at line 14; sequence execution initiated.
INPUT?                                        <- PAUSE command suspends sequence execution.
?read,test1                                   <- Initiate execution of commands in TEST1.

?go                                           <- Resume sequence execution.

 X=10. 20.5 6. 34. 4.4  Y=10.1 21.1 6.1 32.9 4.5
CORRELATION COEFFICIENT IS   1.00
*T #17, END IN L.36
?
     STOP
      1.347 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Debug Session for Second Test Case:

```
CYBER INTERACTIVE DEBUG
?read,bpfile                                  <- Establish breakpoint definitions stored in BPFILE.

?go

*B #1, AT L.14
INPUT?
?read,test2                                   <- Initiate execution of commands in TEST2.

?go

 X=1. 5. 10. 2. 7.  Y=2. 5. 10. 2. 7.
CORRELATION COEFFICIENT IS    .99
*T #17, END IN L.36
?
     STOP
      1.215 CP SECONDS EXECUTION TIME
quit

  DEBUG TERMINATED
```

Figure 5-18. Debug Session Using Command Sequence for Debugging Program CORR (Sheet 1 of 2)

Debug Session for Third Test Case:

```
CYBER INTERACTIVE DEBUG
?read,bpfile
?go

*B #1, AT L.14
INPUT?
?read,test3
?go

 X=2. 2. 2. 2. 0.  Y=1. 5.1 7.6 10. 0.
*B #2, AT L.33
DENOM IS 0
?quit

  DEBUG TERMINATED
```

Establish breakpoint definitions stored in BPFILE.

Initiate execution of commands in TEST3.

Figure 5-18. Debug Session Using Command Sequence for Debugging Program CORR (Sheet 2 of 2)

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,p.newt_l.400 [
 IN COLLECT MODE
? fx=3.0*x-(x+1.0)/(x-1.0)
? print*,"iteration ",its," fx=",fx
? if(fx.le..0001)pause,"execution suspended,check fx"
? if(its.ge.100)pause,"too many iterations"
? go,l.420
? ]
 END COLLECT
? go
  ITERATION 0   FX=1.
  ITERATION 1   FX=6.66666666666664E-02
  ITERATION 2   FX=1.90323320339409E-02
  ITERATION 3   FX=5.67192595682897E-03
  ITERATION 4   FX=1.71038291327363E-03
  ITERATION 5   FX=5.17567172241939E-04
  ITERATION 6   FX=1.56781257640404E-04
  ITERATION 7   FX=4.75071336367705E-05
 *B #1, AT P.NEWT_L.400
 EXECUTION SUSPENDED,CHECK FX
? go
  ITERATION 8   FX=1.43967704246961E-05
 *B #1, AT P.NEWT_L.400
 EXECUTION SUSPENDED,CHECK FX
? execute,p.main_l.150
 CONVERGENCE IN **** ITERATIONS.  X=   0.
 *T #17, END IN P.MAIN_L.170
? quit

SRU     22.400 UNTS.

 RUN COMPLETE.
```

Set breakpoint with body at line 400. Include commands to calculate FX, test for convergence, test number of iterations, and resume execution at line 420.

Initiate execution. The breakpoint body is executed on each pass through the loop, until the IF test is satisfied and the PAUSE command suspends execution of the sequence.

Resume execution of main program.

Figure 5-19. Debug Session Using Command Sequence for Debugging Subroutine NEWT

DEBUGGING IN AN OVERLAY ENVIRONMENT 6

Programs containing overlays can be executed under CID control using all the techniques presented thus far in this manual. In addition, CID provides the following features to facilitate debugging of overlays:

- Address qualification by overlay which allows you to reference locations in different overlays
- An OVERLAY trap that suspends program execution when an overlay is loaded
- Special command forms that limit the command scope to specified overlays

An important fact to remember when debugging programs containing overlays is that while all CID commands are valid for overlays currently in memory, only certain commands can reference locations in overlays that are not loaded.

SUMMARY OF OVERLAY PROCESSING

Overlaying allows you to divide a program into sections called overlays to reduce the amount of memory required for execution. Different overlays can occupy the same storage locations at different times. Thus, when an overlay residing in memory is not currently required by the program, it can be replaced by another overlay.

There are three levels of overlays: a zero level, a primary level, and a secondary level. The zero level, which is sometimes referred to as the main overlay, is resident in memory throughout program execution. The primary level is called from the zero level and is loaded immediately above the zero level. The secondary level is called from its associated primary level or from the zero level and is loaded immediately above the primary level.

A primary level overlay can have any number of secondary level overlays associated with it. When a primary overlay is called from the zero level overlay, it replaces the primary overlay currently residing in memory. When a secondary level overlay is called from a zero level or primary level it replaces the secondary overlay currently residing in memory. Thus, only the zero level, one primary level, and one secondary level can reside in memory concurrently.

Overlays are identified by a pair of integers as follows:

(0,0)	Zero or main overlay
(n,0)	Primary overlay
(n,k)	Secondary overlay

where n is the primary level number and k is the secondary level number. For example, (1,0) and (2,0) are primary overlays; (2,1), (2,2), and (2,3) are secondary overlays associated with primary overlay (2,0).

A group of program units to be loaded into an overlay must be preceded by an OVERLAY directive of the form:

OVERLAY(lfn,i,j)

where lfn is the name of the file on which the overlay is to be written, and i and j are level numbers. The OVERLAY directive must begin in column 7.

Overlays are called from within a FORTRAN program by the statement:

CALL OVERLAY(lfn,i,j)

where lfn is the name of the file in H format on which the overlay is written, and i and j are level numbers.

An example of a program containing overlays is illustrated in figure 6-1. This program contains a main overlay (0,0), two primary level overlays (1,0) and (2,0), and a secondary overlay (1,1) associated with overlay (1,0). The overlays are stored on a file named OVLF, as established by the OVERLAY directives. Each overlay contains a program unit that performs a simple computation. The variables X, Y, and Z are declared in common and are therefore global to the program. The variable RESULT is referenced in three program units and is local to each. The program units in both primary overlays have the same name.

```
      OVERLAY(OVLF,0,0)
      PROGRAM SETXYZ
      COMMON /ACOM/X,Y,Z
      X=1.0
      Y=2.0
      Z=3.0
      CALL OVERLAY(4HOVLF,1,0)
      CALL OVERLAY(4HOVLF,2,0)
      STOP
      END
C
      OVERLAY(OVLF,1,0)
      PROGRAM COMP
      COMMON /ACOM/X,Y,Z
      RESULT=-3.0*X-2.0*Y+2.0*Z
      CALL OVERLAY(4HOVLF,1,1)
      RETURN
      END
C
      OVERLAY(OVLF,1,1)
      PROGRAM COMP2
      COMMON /ACOM/X,Y,Z
      RESULT=5.0*X-6.0*Y+4.0*Z
      RETURN
      END
C
      OVERLAY(OVLF,2,0)
      PROGRAM COMP
      COMMON /ACOM/X,Y,Z
      RESULT=4.0*X+2.0*Y-Z
      RETURN
      END
```

Figure 6-1. Sample Program Illustrating Overlays

The program calculates a value for the variable RESULT in each overlay. Each calculation is local to the overlay in which it resides. The (0,0) overlay is loaded first and remains in memory throughout execution. This overlay sets values for X, Y, and Z and then calls the (1,0) overlay. The (1,0) overlay calculates RESULT, then calls the (1,1) overlay which calculates its local RESULT. At this point, the three overlay levels reside in memory concurrently. When execution of the (1,1) overlay has completed, control returns to the main overlay which then calls the second primary overlay (2,0). Overlay (2,0) replaces overlays (1,0) and (1,1) in memory.

Refer to the Loader reference manual or to the FORTRAN Extended reference manual for more information on overlays.

ADDRESS QUALIFICATION

The address forms presented in tables 2-1 and 2-2 are valid when applied to programs containing overlays. However, when referencing an address in a program unit having the same name as a program unit in another overlay, you must specify the overlay. This is accomplished by prefixing the address specification with an overlay qualifier, as in the following examples:

SET,TRAP,RJ,(2,0)P.MTADD
Set an RJ trap in program unit MTADD residing in overlay (2,0).

SET,BREAKPOINT,(2,1)P.MTADD.L.5
Set a breakpoint at line 5 of program unit MTADD residing in overlay (2,1).

DISPLAY,(0,0)P.SUB1.X
Display the contents of X in program unit SUB1 residing in the zero level overlay.

The following restrictions apply to the use of overlay qualification notation:

- It is necessary to use overlay qualification only when duplicate program unit names exist.
- Overlay qualification cannot be used with the PRINT, IF, and assignment commands.
- If the overlay designation is omitted from an address specification and duplicate program unit names exist, the name of the program unit currently in memory is selected. If none is in memory, the program unit name residing in the overlay having the lowest primary level number is selected.

Overlay qualification notation is also used in CID output messages to denote a particular overlay, as in the trap message:

*T #17, END IN (0,0)P.SETXY.L.9

An END trap has occurred at line 9 of program SETXY in overlay (0,0).

REFERENCING ADDRESSES IN UNLOADED OVERLAYS

The three basic functions of CID, discussed in section 3, have the following restrictions when applied to unloaded overlays:

- Suspending program execution. You can set traps and breakpoints in any overlay, even if it has not been loaded. These traps and breakpoints will be recognized when the containing overlay is loaded and its programs are executed. Program execution cannot be resumed at a location in an unloaded overlay.
- Displaying the contents of program locations. You cannot display the contents of locations within overlays that are not currently loaded. The LIST,VALUES command lists only those variables declared in loaded overlays. LIST,VALUES displays variables in alphabetical order, grouped as to the program unit in which they are defined. Each program unit name is prefixed by an (i,j) overlay qualifier.
- Altering the contents of program locations. You cannot alter the contents of locations within overlays that are not currently loaded.

The following commands can reference addresses in unloaded overlays:

SET,TRAP
SET,BREAKPOINT
SET,HOME
LIST,TRAP
LIST,BREAKPOINT
LIST,MAP
CLEAR,TRAP
CLEAR,BREAKPOINT
SAVE,TRAP
SAVE,BREAKPOINT

If you illegally reference an address in an unloaded overlay, CID issues the error message:

*ERROR - ADDRESS IN UNLOADED OVERLAY

OVERLAY TRAP

The OVERLAY trap suspends program execution and gives control to CID whenever specified overlays are loaded into memory. This allows you to examine and alter the status of a program as it exists at the time the overlay is loaded. The trap occurs after the overlay is loaded but before control transfers to the loaded overlay. The forms of the command to set an overlay trap are:

SET,TRAP,OVERLAY,*
CID gets control when any overlay is loaded.

SET,TRAP,OVERLAY,(i,j)
CID gets control when overlay (i,j) is loaded.

When an overlay trap occurs, CID issues the message:

T #n, OVERLAY (i,j) IN (i,j)P.name.L.0

n	Trap number assigned by CID
i,j	Level numbers of the current overlay
name	Name of the program unit in the current overlay to be executed first

SPECIAL FORMS OF SOME DEBUG COMMANDS

The commands to LIST, CLEAR, and SAVE traps and breakpoints, and the LIST,MAP command have special forms intended for use with overlay programs. These forms, listed in table 6-1, allow you to specify an overlay or list of overlays for the scope parameter.

The LIST,MAP command is especially useful when used with overlay programs. This command lists all program modules grouped according to overlay. Overlays are identified by an (i,j) designation. Overlays currently in memory are indicated by an asterisk.

OVERLAY EXAMPLE

A debug session for the program in figure 6-1 is illustrated in figure 6-2. The purpose of this session is to suspend program execution after each overlay is loaded and to issue various commands to examine the status of the program. Control statements are included to activate debug mode and compile, load, and execute the program. CID must be on at compile time and at the time the load sequence is issued.

The following CID commands are issued during this session:

LIST,MAP
: List the level numbers of all overlays in the program. Overlays currently in memory are indicated by an asterisk.

LIST,MAP,(2,0)
: List program modules contained in the (2,0) overlay.

SET,TRAP,OVERLAY,*
: Set an overlay trap that suspends execution when any overlay is loaded.

TRACEBACK
: Display a program traceback list starting at the home program.

DISPLAY,#HOME
: Display the name of the home program and the overlay in which it resides.

SET,HOME,(1,0)P.COMP
: Designate COMP in overlay (1,0) as the home program. Note that the (1,0) overlay is not in memory when this command is issued. Variables declared in COMP cannot be referenced, as illustrated by the next command.

PRINT*,RESULT
: Attempt to print the value of RESULT. The attempt fails since the overlay containing the current home program is not in memory.

TABLE 6-1. SPECIAL FORMS OF CID COMMANDS FOR OVERLAY PROGRAMS

Command	Description
LIST,TRAP,type,(i,j),(i,j),...	Lists addresses of traps in the specified overlays; * can be substituted for type, in which case all types are listed.
LIST,MAP,(i,j),(i,j),...	Lists program modules contained in the specified overlays.
LIST,BREAKPOINT,(i,j),(i,j),...	Lists addresses of all breakpoints in the specified overlays.
CLEAR,TRAP,type,(i,j),(i,j),...	Clears all traps of the specified type in the specified overlays.
CLEAR,BREAKPOINT,(i,j),(i,j),...	Clears all breakpoints in the specified overlays.
SAVE,BREAKPOINT,lfn,(i,j),(i,j),...	Copies the breakpoints in the specified overlays to lfn.
SAVE,TRAP,lfn,type,(i,j),(i,j),...	Copies the traps of the specified type in the specified overlays to lfn; * can be substituted for type, in which case all types are copied.

```
..debug

..ftn,i=ovprog,ts,db

   41000B   CM STORAGE USED
       .173 CP SECONDS COMPILATION TIME
..xeq
OPTION=load=lgo   <-- Load sequence.
OPTION=nogo

..ovlf

CYBER INTERACTIVE DEBUG
?list,map

(0,0) * ,   (1,0),   (1,1),   (2,0)   <-- Overlay (0,0) is currently in memory.
?set,trap,overlay,*   <-- Set OVERLAY trap; scope is entire program.

?go

*T #1, OVERLAY (1,0) IN (1,0)P.COMP_L.0   <-- OVERLAY trap occurs when overlay (1,0) is loaded.
?list,map

(0,0) * ,   (1,0) * ,   (1,1),   (2,0)   <-- Overlays (0,0) and (1,0) are currently in memory.
?go

*T #1, OVERLAY (1,1) IN (1,1)P.COMP2_L.0   <-- OVERLAY trap occurs when overlay (1,1) is loaded.
?list,map

(0,0) * ,   (1,0) * ,   (1,1) * ,   (2,0)   <-- Overlays (0,0), (1,0) and (1,1) are currently in
?display,#home                                  memory.

#HOME = (1,1)P.COMP2   <-- Program COMP2 in overlay (1,1) is the current
?list,values                home program.

(0,0)P.SETXYZ                 <-- List all variables and values in programs currently
X = 1.0,   Y = 2.0,   Z = 3.0     in memory.
(1,0)P.COMP
RESULT = -1.0,   X = 1.0,   Y = 2.0,   Z = 3.0
(1,1)P.COMP2
RESULT = -I,   X = 1.0,   Y = 2.0,   Z = 3.0
?traceback   <-- Initiate traceback from current home program.

P.COMP2 CALLED FROM P.COMP_L.6
P.COMP CALLED FROM P.SETXYZ_L.7
?go

*T #1, OVERLAY (2,0) IN (2,0)P.COMP_L.0   <-- OVERLAY trap occurs when overlay (2,0) is loaded.
?list,map

(0,0) * ,   (1,0),   (1,1),   (2,0) *   <-- Overlays (0,0) and (2,0) are currently in memory.

?go

*T #17, END IN (0,0)P.SETXYZ_L.9   <-- Program terminates at line 9 of program SETXYZ
?                                      in overlay (0,0).
     STOP
      1.522 CP SECONDS EXECUTION TIME
list,map

(0,0) * ,   (1,0),   (1,1),   (2,0) *   <-- Overlays (0,0) and (2,0) are currently in memory.
?list,values   <-- List all variables and values in programs currently
                   in memory.
(0,0)P.SETXYZ
X = 1.0,   Y = 2.0,   Z = 3.0
(2,0)P.COMP
RESULT = 5.0,   X = 1.0,   Y = 2.0,   Z = 3.0
```

Figure 6-2. Debug Session Illustrating Overlay Debugging (NOS/BE) (Sheet 1 of 2)

```
?print*,result

*ERROR - NO PROGRAM VARIABLE RESULT <-------- RESULT is not defined in current home program.
?set,home,(1,0)p.comp <-------- Designate program COMP in overlay (1,0) as
                                home program.

?print*,result

*ERROR - ADDRESS IN UNLOADED OVERLAY <-------- Current home program is in an unloaded overlay.
?set,home,(2,0)p.comp <-------- Designate program COMP in overlay (2,0) as
                                home program.

?print*,result

 5.
?quit

  DEBUG TERMINATED
```

Figure 6-2. Debug Session Illustrating Overlay Debugging (NOS/BE) (Sheet 2 of 2)

STANDARD CHARACTER SETS

Control Data operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

The set in use at a particular installation is specified when the operating system is installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect throughout the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1; 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Graphic character representation appearing at a terminal or printer depends on the installation character set and the terminal type. Characters shown in the CDC Graphic column of the standard character set table are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

Display Code (octal)	CDC			ASCII		
	Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
00†	: (colon)††	8-2	00	: (colon) ††	8-2	072
01	A	12-1	61	A	12-1	101
02	B	12-2	62	B	12-2	102
03	C	12-3	63	C	12-3	103
04	D	12-4	64	D	12-4	104
05	E	12-5	65	E	12-5	105
06	F	12-6	66	F	12-6	106
07	G	12-7	67	G	12-7	107
10	H	12-8	70	H	12-8	110
11	I	12-9	71	I	12-9	111
12	J	11-1	41	J	11-1	112
13	K	11-2	42	K	11-2	113
14	L	11-3	43	L	11-3	114
15	M	11-4	44	M	11-4	115
16	N	11-5	45	N	11-5	116
17	O	11-6	46	O	11-6	117
20	P	11-7	47	P	11-7	120
21	Q	11-8	50	Q	11-8	121
22	R	11-9	51	R	11-9	122
23	S	0-2	22	S	0-2	123
24	T	0-3	23	T	0-3	124
25	U	0-4	24	U	0-4	125
26	V	0-5	25	V	0-5	126
27	W	0-6	26	W	0-6	127
30	X	0-7	27	X	0-7	130
31	Y	0-8	30	Y	0-8	131
32	Z	0-9	31	Z	0-9	132
33	0	0	12	0	0	060
34	1	1	01	1	1	061
35	2	2	02	2	2	062
36	3	3	03	3	3	063
37	4	4	04	4	4	064
40	5	5	05	5	5	065
41	6	6	06	6	6	066
42	7	7	07	7	7	067
43	8	8	10	8	8	070
44	9	9	11	9	9	071
45	+	12	60	+	12-8-6	053
46	-	11	40	-	11	055
47	*	11-8-4	54	*	11-8-4	052
50	/	0-1	21	/	0-1	057
51	(	0-8-4	34	(	12-8-5	050
52	)	12-8-4	74	)	11-8-5	051
53	$	11-8-3	53	$	11-8-3	044
54	=	8-3	13	=	8-6	075
55	blank	no punch	20	blank	no punch	040
56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
57	. (period)	12-8-3	73	. (period)	12-8-3	056
60	≡	0-8-6	36	#	8-3	043
61	[	8-7	17	[	12-8-2	133
62	]	0-8-2	32	]	11-8-2	135
63	% ††	8-6	16	% ††	0-8-4	045
64	≠	8-4	14	" (quote)	8-7	042
65	←	0-8-5	35	_ (underline)	0-8-5	137
66	∨	11-0 or 11-8-2†††	52	!	12-8-7 or 11-0†††	041
67	∧	0-8-7	37	&	12	046
70	↑	11-8-5	55	' (apostrophe)	8-5	047
71	↓	11-8-6	56	?	0-8-7	077
72	<	12-0 or 12-8-2†††	72	<	12-8-4 or 12-0†††	074
73	>	11-8-7	57	>	0-8-6	076
74	≤	8-5	15	@	8-4	100
75	≥	12-8-5	75	\	0-8-2	134
76	¬	12-8-6	76	~ (circumflex)	11-8-7	136
77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

†Twelve zero bits at the end of a 60-bit word in a zero byte record are an end of record mark rather than two colons.

††In installations using a 63-graphic set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55_8).

†††The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

CDC CHARACTER SET
COLLATING SEQUENCE

Collating Sequence Decimal	Collating Sequence Octal	CDC Graphic	Display Code	External BCD	Collating Sequence Decimal	Collating Sequence Octal	CDC Graphic	Display Code	External BCD
00	00	blank	55	20	32	40	H	10	70
01	01	≤	74	15	33	41	I	11	71
02	02	%	63 †	16 †	34	42	v	66	52
03	03	[	61	17	35	43	J	12	41
04	04	→	65	35	36	44	K	13	42
05	05	≡	60	36	37	45	L	14	43
06	06	^	67	37	38	46	M	15	44
07	07	↑	70	55	39	47	N	16	45
08	10	↓	71	56	40	50	O	17	46
09	11	>	73	57	41	51	P	20	47
10	12	≥	75	75	42	52	Q	21	50
11	13	¬	76	76	43	53	R	22	51
12	14	.	57	73	44	54	]	62	32
13	15	)	52	74	45	55	S	23	22
14	16	;	77	77	46	56	T	24	23
15	17	+	45	60	47	57	U	25	24
16	20	$	53	53	48	60	V	26	25
17	21	*	47	54	49	61	W	27	26
18	22	-	46	40	50	62	X	30	27
19	23	/	50	21	51	63	Y	31	30
20	24	,	56	33	52	64	Z	32	31
21	25	(	51	34	53	65	:	00 †	none†
22	26	=	54	13	54	66	0	33	12
23	27	≠	64	14	55	67	1	34	01
24	30	<	72	72	56	70	2	35	02
25	31	A	01	61	57	71	3	36	03
26	32	B	02	62	58	72	4	37	04
27	33	C	03	63	59	73	5	40	05
28	34	D	04	64	60	74	6	41	06
29	35	E	05	65	61	75	7	42	07
30	36	F	06	66	62	76	8	43	10
31	37	G	07	67	63	77	9	44	11

†In installations using the 63-graphic set, the % graphic does not exist. The : graphic is display code 63, External BCD code 16.

GLOSSARY B

Abort -
To terminate a program or job when an error condition (hardware or software) exists from which the program or computer cannot recover.

Auxiliary File -
An optional file, established by the SET,AUXILIARY command, to which CID output is written. The output types written to this file are specified by special output codes.

Batch Mode -
A mode of CID execution which allows programs intended for batch execution to be executed under CID control.

Breakpoint -
A designated location in a program where execution is to be suspended.

Collect Mode -
A mode of CID execution in which commands entered by the user are not executed, but are included in a group, trap, or breakpoint body; initiated by a at the end of a SET,TRAP; SET,GROUP; or SET,BREAKPOINT command and terminated by a .

Common Block -
A module intended solely for storing data. A block of data can be declared in common to both the calling routine and the called routine as an alternative to passing data to routines via parameter values.

Debug Mode -
A mode of execution in which special CID tables are generated during compilation and in which user programs are executed under CID control; initiated by a DEBUG(ON) control statement and terminated by a DEBUG(OFF) control statement.

Debug Session -
A sequence of interactions between the user and CID, beginning when execution of the user program is initiated in debug mode and ending when a QUIT command is issued.

Entry Point -
A special named location within a program module. This location is, by convention, the target of the RJ (Return Jump) instruction that transfers control to the module. The RJ instruction stores the return address at the entry point location and starts execution at the next location.

Extended Core Storage (ECS) -
An additional memory available as an option on CYBER computer systems. This memory can only be used for program and data storage, not for program execution. Special hardware instructions exist for transferring data between central memory and ECS.

Group -
A sequence of CID commands established and assigned a name by a SET,GROUP command and executed when a READ command is issued.

Home Program -
Program unit in which variables, line numbers, and statement labels referenced by the user in CID commands are assumed to be located unless appropriate qualifiers appear. By default, the home program is the program unit being executed at the time CID gains control. The user can change the default with the SET,HOME command.

Interactive -
Capable of a two-way back and forth exchange of information.

Interactive Mode -
The normal mode of CID execution. The user enters commands directly from the terminal and CID immediately executes the commands. CID can also execute in batch mode.

Interpret Mode -
A mode of execution in which a special routine called an interpreter examines each machine instruction to be executed in the user program, and simulates its execution by the execution of several of its own instructions. Execution in interpret mode consequently takes 20 to 50 times as long as direct execution.

Interrupt (verb) -
To stop a running program in such a way that it can be resumed at a later time.

Interrupt (noun) -
A special control signal which, when issued, causes action as described for INTERRUPT (verb).

Module -
A named section of coding output by a compiler or assembler, or a block of data (common block). The components of system libraries are also modules. Prior to loading, modules are called object modules; after loading they are called load modules.

Optimizing Mode -
One of the compilation modes of the FORTRAN Extended compiler, indicated by the control statement options OPT=0, 1, 2, or by omission of the TS option. Programs compiled in optimizing mode cannot use many of the CID features described in this manual.

Overlay -
A portion of a program, consisting of one or more modules, which can share an allocated area of memory with other portions of the program. When access to a particular module is required, the overlay containing that module is loaded, thus overlaying the previous contents of the memory area allocated for that overlay. Such a scheme allows large programs to execute in a limited amount of memory.

Program -
The completely loaded set of one or more object modules. A program that has been loaded in debug mode can be executed under CID control.

Program Module -
A module intended for program execution. A program module always has an entry point, a named location in the module to be used in calling the module via the RJ instruction.

Program Unit -
A FORTRAN main program, function subprogram, subroutine, or block data subprogram.

Time-Sharing Mode -
One of the compilation modes of the FORTRAN Extended compiler, indicated by the TS control statement option; required for full use of features described in this manual.

Trap (verb) -
To suspend program execution and transfer control to CID upon the detection of some specified condition.

Trap (noun) -
A mechanism that detects the occurrence of a specified condition, suspends execution of the user program at that point, and transfers control to CID.

Veto Mode -
A mode of CID execution in which CID displays each command of a command sequence immediately prior to its execution and gives control to the user. The user can allow the command to be executed, skipped, or replaced by another command. Veto mode is activated by the SET,VETO,ON command and terminated by the SET,VETO,OFF or CLEAR,VETO command.

The following paragraphs discuss some of the errors that can cause a FORTRAN program to terminate prematurely. To use CID effectively, you should be able to recognize situations that can cause these errors. A knowledge of the internal representation of numbers can be helpful in understanding why arithmetic errors occur.

FLOATING POINT REPRESENTATION

The internal floating point format is shown in figure C-1. Bits 0 through 47 contain the coefficient of the number, equivalent to about 14 decimal digits. The binary point is assumed to be at the right of bit 0. The sign of the coefficient is represented by bit 59, 0 for a positive value and 1 for a negative value. The exponent is contained in bits 48 through 58 and is biased by 2000 octal, that is, 2000 octal is added to the exponent. Some examples of internal floating point representation, including the largest and smallest permissible values, are illustrated in table C-1. Operands exceeding the maximum or minimum values cause execution errors.

ARITHMETIC MODE ERRORS

Arithmetic mode errors occur when the central processor encounters an instruction that cannot be executed. Such an instruction generally involves an operand that contains invalid data or an address beyond the user's field length. The arithmetic mode errors and some possible causes are listed in table C-2.

TABLE C-1. EXAMPLES OF FLOATING POINT NUMBERS

Number	Internal Representation
+1.	1720 4000 0000 0000 0000
+100.	1726 6200 0000 0000 0000
-100.	6051 1577 7777 7777 7777
1.E64	2245 6047 4037 2237 7733
-1.E64	6404 2570 0025 6605 5317
0.	0000 0000 0000 0000 0000

When a mode error occurs, the executing program is aborted and a message of the following format is issued:

time ERROR MODE=n ADDRESS=xxxxxx

where n is the mode number and xxxxxx is the relative octal address where the error occurred.

When a mode error occurs while executing in debug mode, control passes to CID. This is due to the ABORT trap explained in section 3. On receiving control, CID issues the following message and prompt:

*T#18 ABORT CPU ERROR EXIT n IN L.m
?

where n is the mode number and m is the source line number where the error occurred. You can then enter CID commands to examine the status of the program as it exists at the time of termination.

OVERFLOW, UNDERFLOW, DIVISION BY ZERO

Floating point overflow occurs when a value is generated which exceeds the allowable exponent range. This situation can occur when performing calculations with numbers of extremely large magnitude or when a nonzero number is divided by zero.

When overflow occurs, the exponent is set to 37777_8 (the largest possible exponent) and the characteristic is set to zero. Such an operand is called an infinite operand. The executing program aborts when the infinite operand is used in a subsequent computation, not when it is generated. A debug session for a program that generates an infinite operand is illustrated in figure C-2. A division by zero generates the infinite operand. Program execution terminates when the infinite operand is referenced in the statement D=C+1.0. The LIST,VALUES command shows the program variables as they exist at the time of termination. The value of the variable C, the infinite operand, is represented by the letter R.

Floating point underflow occurs when a value is generated which would have an exponent less than -294. The resulting operand is set to all zeros.

The allowable range for floating point numbers is shown in table C-3.

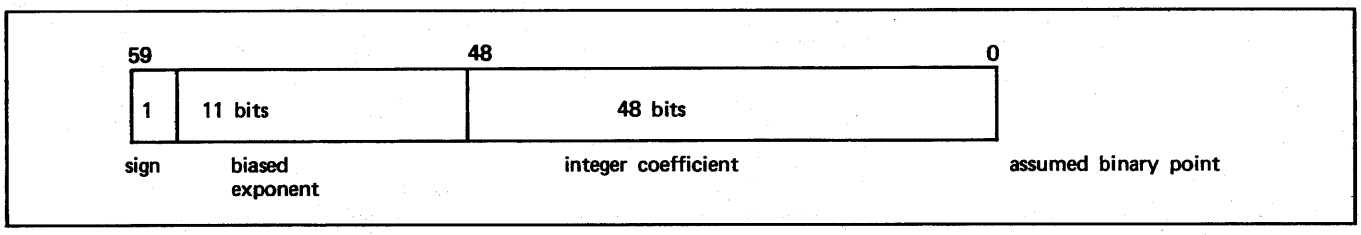

Figure C-1. Internal Floating Point Format

TABLE C-2. MODE ERRORS

Error Number	Explanation	Possible Causes
0	A branch to location zero has occurred or an illegal instruction has been executed.	Machine instructions destroyed by array overflow; mismatch between number of arguments in CALL and SUBROUTINE statement.
1	Program has referenced a location outside the user's field length.	a. Program attempted to fetch or store data outside program field length; probably due to incorrect subscript. b. Program attempted to jump to an address outside program field length; probably due to missing function or subroutine or misspelled function or subroutine name. c. Program executed a word that did not contain an instruction; probably due to array overflow.
2	Infinite value used as operand.	Nonzero number divided by zero; very large number divided by very small number; very large number multiplied by very large number; numbers very near 10^{322} in absolute value added or subtracted; large number raised to a large power.
4	Indefinite value used as operand.	Undefined value used in a calculation; zero divided by zero; zero multiplied by an infinite value; infinite value divided by an infinite value; infinite values added or subtracted.
3	Combination of 1 and 2.	
5	Combination of 1 and 4.	
6	Combination of 2 and 4.	
7	Combination of 1, 2, and 4.	

Program Listing:

```
1       PROGRAM COMPC        74/74   TS ID                          FTN

                             PROGRAM COMPC
                             A=1.0
                             B=1.0
                             C=(A+B)/(A-B)
      5                      D=C+1.0
                             STOP
                             END
```

Session Log:

```
1                                                CYBER INTERACTIVE DEBUG
0
    EXECUTE
    *T =18, ABORT CPU ERROR EXIT 02 IN L.5
    LIST,VALUES
    P.COMPC
    A = 1.0,  B = 1.0,  C = R,  D = R
    QUIT

        DEBUG TERMINATED
```

Figure C-2. Program COMPC and Debug Session Illustrating Arithmetic Error

TABLE C-3. FLOATING POINT REPRESENTATIONS

	Positive Operand		Negative Operand	
	Floating Point Representation	Octal	Floating	Octal
Largest Value	≅ 1.265014083171E+322	$37767...7_8$	≅ -1.265014083171E+322	$40010...0_8$
Smallest Value	≅ 3.131513062514E-294	$000140...0_8$	≅ -3.131513062514E-294	$777637...7_8$
Zero (underflow yields zero operand)	0.0	$0...0_8$	-0.0	$7...7_8$
Overflow (infinite operand)	R	$37770...0_8$	-R	$4000...0_8$
Indefinite	I	$17770...0_8$	-I	$60007...7_8$

INDEFINITE OPERANDS

Indefinite operands are generated when the central processor encounters an instruction that cannot be resolved, such as a division where both the dividend and the divisor have a value of zero. Indefinite operands can also be generated when a variable has not been initialized (the value assigned to uninitialized areas of memory is an installation parameter). As with infinite operands, indefinites cause abnormal termination of execution when they are referenced.

An indefinite operand is represented by the character I when displayed. The internal representation of an indefinite operand is shown in table C-3.

ERRORS INVOLVING INTEGERS

The maximum permissible absolute value of an integer depends on the context in which it is used. When an integer exceeds the limits of the central processor, it is assigned a value of zero. The allowable range for integers is shown in table C-4.

TABLE C-4. INTEGER REPRESENTATIONS

	Positive Operand		Negative Operand	
	Integer	Octal	Integer	Octal
Maximum value for arithmetic operations	$2^{48}-1$ (5761 7927 7326 7128 31)	$37767...7_8$	$-(2^{48}-1)$	$40010...0_8$
Maximum value for subscripts and DO loop index	$2^{17}-1$ (131071)	$0...037777_8$	Negative operand not permitted	
Zero	0	$0...0$	-0	$7...7_8$
Infinite Operand		Set to zero		
Indefinite Operand		Set to zero		

BATCH MODE DEBUGGING

CID is primarily intended to be used interactively, but can be used in batch mode. Possible reasons for using batch mode include the possibility of a large volume of output or lack of access to a terminal. In batch mode, however, you must plan the entire session in advance. This requires care and can also require a knowledge of what errors are likely to occur.

To conduct a debug session in batch mode, commands must exist on a file of card images called DBUGIN from which CID reads all input. You can create this file using the system text editor or you can punch the commands on cards and include them as part of the job deck, copying file INPUT to DBUGIN. Commands are punched or written in the same format as in interactive mode; a card can contain a single command or multiple commands separated by semicolons.

As in interactive mode, debug mode is established by the DEBUG control statement. The debug session is initiated by a statement to load and execute the program. Control transfers immediately to CID, which begins executing the commands in DBUGIN. When CID encounters a GO or EXECUTE in the command stream, control transfers to the user program. The user program executes until a trap or breakpoint is encountered. In this manner, control transfers between the program and CID with no user intervention.

A QUIT command is normally the last command of the sequence. However, this command can be omitted and CID will terminate after the last command has been executed.

Following are some restrictions that apply to batch mode debugging:

- Invalid commands are disregarded; when CID encounters such a command, processing continues with the next command.
- Commands that would generate a warning message in interactive mode are executed in batch mode.
- You cannot establish veto mode in a batch session; CID executes all commands except when execution is impossible.

All output from CID is written to a file named DBUGOUT in batch mode. This is a local file and it is the user's responsibility to print the file or make it permanent. You can control the types of output sent to DBUGOUT with the SET,OUTPUT command. Output can also be sent to a separate file with the SET,AUXILIARY command.

Batch output from a debug session does not normally show the user-specified CID commands as they are executed. CID reads the commands from DBUGIN but does not echo them to DBUGOUT unless the T option is specified on the SET,OUTPUT command. Use of this option usually improves the readability of a batch debug session.

With the exception of the SET,VETO command, all CID commands are valid in batch mode. You can set traps and breakpoints, define command sequences, display and alter the values of program variables, and resume program execution. The commands in DBUGIN should be specified in the same order as in interactive mode. CID accesses DBUGIN for all input that would normally be input from the terminal.

An example of a program to be debugged in batch mode is illustrated in figure D-1. Breakpoints are set initially at lines 2 and 6, and program execution is initiated. When the first breakpoint is encountered, CID receives control and executes commands until the next GO is encountered. The command GO,L.5 skips the FORTRAN statements that rewind and read an input file and test for end-of-file. When the breakpoint at line 6 is encountered, CID executes the LIST,VALUES and QUIT commands. The contents of the output file DBUGOUT are shown in figure D-2.

Deck setup for NOS/BE:

```
JOB statement
COPYBR(INPUT,DBUGIN)
REWIND(DBUGIN)
DEBUG(ON)
FTN.
LGO.
CATALOG(DBUGOUT,ID=MYID,RP=100)
7/8/9 in column 1
SET,BREAKPOINT,L.2
SET,BREAKPOINT,L.6
GO
X1=0.0;Y1=0.0
X2=2.0;Y2=0.0
X3=1.0;Y3=-1.0
GO,L.5
LIST,VALUES,P.RD
QUIT
7/8/9 in column 1
      PROGRAM RD(INPUT,OUTPUT,TRFILE,TAPE2=TRFILE)
      REWIND 2
   10 READ(2,*) X1,Y1,X2,Y2,X3,Y3
      IF(EOF(2).NE.0) GO TO 20
      CALL AREA(X1,Y1,X2,Y2,X3,Y3,A)
      GO TO 10
   20 STOP
      END
      SUBROUTINE AREA(X1,Y1,X2,Y2,X3,Y3,A)
      S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
      S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
      S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
      T=(S1+S2+S3)/2.0
      A=SQRT(T*(T-S1)*(T-S2)*(T-S3))
      RETURN
      END
6/7/8/9 in column 1
```

When the session is initiated these commands are executed.

Transfer control to user program.

When the breakpoint at line 2 is encountered these commands are executed.

Transfer control to line 5 of user program.

When the breakpoint at line 6 is encountered these commands are executed.

Deck setup for NOS:

```
JOB statement
ACCOUNT statement
COPYBR(INPUT,DBUGIN)
REWIND(DBUGIN)
DEFINE(DBUGOUT)
DEBUG(ON)
FTN.
LGO.
7/8/9 in column 1
SET,BREAKPOINT,L.2
SET,BREAKPOINT,L.5
GO
X1=0.0;Y1=0.0
X2=2.0;Y2=0.0
X3=1.0;Y3=-1.0
GO,L.5
LIST,VALUES,P.RD
QUIT
7/8/9 in column 1
```

When the session is initiated these commands are executed.

Transfer control to user program

When the breakpoint at line 2 is encountered these commands are executed.

Transfer control to line 5 of user program.

When the breakpoint at line 6 is encountered these commands are executed.

Figure D-1. Sample Job Deck for Batch Mode Debugging (Sheet 1 of 2)

```
      PROGRAM RD(INPUT,OUTPUT,TRFILE,TAPE2=TRFILE)
      REWIND 2
   10 READ(2,*) X1,Y1,X2,Y2,X3,Y3
      IF(EOF(2).NE.0) GO TO 20
      CALL AREA(X1,Y1,X2,Y2,X3,Y3,A)
      GO TO 10
   20 STOP
      END
      SUBROUTINE AREA(X1,Y1,X2,Y2,X3,Y3,A)
      S1=SQRT((X2-X1)**2 + (Y2-Y1)**2)
      S2=SQRT((X3-X1)**2 + (Y3-Y1)**2)
      S3=SQRT((X3-X2)**2 + (Y3-Y2)**2)
      T=(S1+S2+S3)/2.0
      A=SQRT(T*(T-S1)*(T-S2)*(T-S3))
      RETURN
      END
6/7/8/9 in column 1
```

Figure D-1. Sample Job Deck for Batch Mode Debugging (Sheet 2 of 2)

```
                        CYBER INTERACTIVE DEBUG

1
0
 CYBER INTERACTIVE DEBUG
 SET,BREAKPOINT,L.2
 SET,BREAKPOINT,L.6
 GO
 *B #1, AT L.2
 X1=0.0;Y1=0.0
 X2=2.0;Y2=0.0
 X3=1.0;Y3=-1.0
 GO,L.5
 *B #2, AT L.6
 LIST,VALUES,P.RD
 P.RD
 A = .99999999999999,   X1 = 0.0,   X2 = 2.0,   X3 = 1.0,   Y1 = 0.0,
 Y2 = 0.0,   Y3 = -1.0
 QUIT
```

Figure D-2. Contents of File DBUGOUT

SUMMARY OF DEBUG COMMANDS

Table E-1 summarizes the Debug commands and specifies the page in this manual where more detailed information can be obtained.

TABLE E-1. CID COMMAND SUMMARY

Command	Short Form	Described on Page	Description
†assignment (var=expr)			The value of the expression on the right of the equal sign replaces the current value of the variable on the left of the equal sign.
CLEAR,AUXILIARY	CAUX		Purges the auxiliary output file.
CLEAR,BREAKPOINT	CB		Removes breakpoints.
CLEAR,GROUP	CG		Removes command group definitions.
CLEAR,INTERPRET	CI		Turns off interpret mode.
CLEAR,OUTPUT	COUT		Turns off output to the terminal.
CLEAR,TRAP	CT		Removes traps.
CLEAR,VETO	CV		Turns off veto mode.
DISPLAY	D		Displays the contents of program locations.
ENTER	E		Stores values into program locations.
EXECUTE	EXEC		Resumes execution of the user program.
GO			Resumes execution of the user program or of a suspended command sequence.
†IF			Provides for conditional execution of CID commands.
JUMP			Causes a transfer of control within a command sequence.
LABEL			Designates a label to be used as the destination of a JUMP command.
LIST,BREAKPOINT	LB		Displays information about breakpoints defined for the current session.
LIST,GROUP	LG		Displays information about command groups defined for the current session.
LIST,MAP	LM		Displays load map information.
LIST,STATUS	LS		Displays information about the current status of the debug session.
LIST,TRAP	LT		Displays information about traps currently defined for the debug session.
LIST,VALUES	LV		Displays names and values of program variables.
MESSAGE			Displays a string of characters.
MOVE	M		Moves data from one location to another.

TABLE E-1. CID COMMAND SUMMARY (Cont'd)

Command	Short Form	Described on Page	Description
PAUSE			Suspends execution of the currently executing command sequence.
†PRINT			Displays the contents of program variables.
QUIT			Terminates the debug session.
READ			Executes a group or file sequence or trap, breakpoint, and group definitions saved on a file.
SAVE,BREAKPOINT	SAVEB		Writes breakpoint definitions to a file.
SAVE,GROUP	SAVEG		Writes group definitions to a file.
SAVE,TRAP	SAVET		Writes trap definitions to a file.
SET,AUXILIARY	SAUX		Establishes an auxiliary output file.
SET,BREAKPOINT	SB		Establishes breakpoints.
SET,HOME	SH		Designates a home program.
SET,OUTPUT	SOUT		Selects output types to be displayed at the terminal.
SET,GROUP	SG		Defines a command group.
SET,INTERPRET,ON	SI ON		Turns on interpret mode.
SET,INTERPRET,OFF	SI OFF		Turns off interpret mode.
SET,TRAP	ST		Establishes traps.
SET,VETO,ON	SV ON		Turns on veto mode.
SET,VETO,OFF	SV OFF		Turns off veto mode.
SKIPIF			Provides for conditional execution of CID commands.
SUSPEND			Suspends the debug session.
TRACEBACK			Lists a subroutine call sequence.

†Valid only with FORTRAN programs compiled in debug mode or with the DB option.

INDEX

COMMENT SHEET

CONTROL DATA CORPORATION

TITLE: CYBER Interactive Debug Version 1 Guide
For Users of FORTRAN Extended Version 4

PUBLICATION NO. 60482700 REVISION A

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

CUT ON THIS LINE

General comments:

FROM NAME: _____ POSITION: _____

COMPANY NAME: _____

ADDRESS: _____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

TAPE

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
P.O. BOX 3492
Sunnyvale, California 94088-3492

CUT ALONG LINE

FOLD

FOLD

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.

CONTROL DATA CORPORATION