60484120

CONTROL DATA CORPORATION

CYBER INTERACTIVE DEBUG VERSION 1 GUIDE FOR USERS OF COBOL VERSION 5

CDC® OPERATING SYSTEMS:

NOS 2

NOS/BE 1

60484120

CONTROL DATA CORPORATION

CYBER INTERACTIVE DEBUG VERSION 1 GUIDE FOR USERS OF COBOL VERSION 5

CDC® OPERATING SYSTEMS:
NOS 2
NOS/BE 1

REVISION RECORD

Revision	Description
A (03/23/82)	Initial release under NOS 2 and NOS/BE 1; PSR level 552.
B (08/22/84)	Revised at PSR level 601 to document support of the CYBER 170 800 Series models and the CYBER 180 Computer Systems. This revision includes clarification of SET,OUTPUT default options and definition of the underscore used in CID.

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
Front Cover	-
Title Page	-
ii	B
iii/iv	B
v	B
vi	A
vii	A
viii	A
1-1	A
1-2	A
2-1 thru 2-8	A
3-1 thru 3-16	A
3-17	B
3-18	A
3-19	B
3-20 thru 3-30	A
4-1 thru 4-10	A
5-1 thru 5-15	A
A-1	A
A-2	A
B-1	A
C-1 thru C-4	A
D-1	A
D-2	A
Index-1	A
Index-2	A
Comment Sheet/Mailer	B
Mailer	-
Back Cover	-

PREFACE

This manual provides the COBOL programmer with assistance in debugging COBOL Version 5 programs under the control of the CDC® CYBER Interactive Debug Facility.

As described in this publication, CYBER Interactive Debug (CID) operates under the following operating systems:

NOS 2 for the CDC CYBER 180 Computer Systems; CYBER 170 Computer Systems; CYBER 70 Computer Systems Models 71, 72, 73, 74; and 6000 Computer Systems

NOS/BE 1 for the CONTROL DATA® CYBER 180 Computer Systems; CYBER 170 Computer Systems; CYBER 70 Computer Systems Models 71, 72, 73, 74; and 6000 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 COBOL 5 and should be able to run jobs interactively under either the NOS Interactive Facility, or NOS/BE INTERCOM.

This guide provides a tutorial approach to CID beginning with basic features and proceeding through more advanced features. Section 1 provides some background information and presents a summary of the features of CID. Section 2 describes the method for initiating a debug session with CID, and describes several useful CID commands; this section contains sufficient information to allow the user to make productive use of CID. Sections 3 through 5 describe features which are helpful in debugging more complex programs. This guide is not comprehensive in its approach to CID; only those features considered useful to COBOL programmers are described.Most of the features described 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.

The Software Publications Release History serves as a guide for determining which revision level of software documentation corresponds to the Programming System Report (PSR) level of installed site software.

The following manuals are of primary interest:

Publication	Publication Number
CYBER Interactive Debug Version 1 Reference Manual	60481400
CID Version 1 Reference Manual Online	L60481400
COBOL Version 5 Reference Manual	60497100
COBOL Version 5 Reference Manual Online	L60497100

The following manuals are of secondary interest:

Publication	Publication Number
INTERCOM Version 5 Reference Manual	60455010
NOS Version 2 Reference Set, Volume 1 Introduction to Interface Usage	60459660
NOS Version 2 Reference Set, Volume 3 System Commands	60459680
Software Publications Release History	60481000
XEDIT Version 3 Reference Manual	60455730

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

CONTENTS

FIGURES

TABLES

INTRODUCTION

CYBER Interactive Debug (CID) Version 1, allows you to interactively debug an executing COBOL 5 program. CID can be used with COBOL 5 programs compiled under the NOS or NOS/BE operating systems.

Use of CID requires a mode of execution called debug mode. Debug mode is turned on by 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 when selected conditions occur, such as reaching the beginning of a line.

Display the values of data-items and tables while execution is suspended.

Change the values of data-items within the program while execution is suspended.

Resume program execution at the location where execution was suspended.

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 within the program.

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 COBOL program several times to make corrections or to add statements that print intermediate program values. These debugging techniques can be expensive in terms of both machine time and programmer time.

CID, however, does not require a knowledge of assembly language or the ability to interpret memory dumps. You can completely debug a program with CID by referring only to a source listing and by referencing data-items and line numbers symbolically. In many cases, a COBOL 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 flow of control as execution proceeds, you can often accomplish, in a single session, debugging that would normally require several compilations. Thus, considerable time can be saved, especially when you are debugging programs that are time-consuming to compile or execute.

SPECIAL CID FEATURES FOR COBOL PROGRAMS

CID provides certain features available only to COBOL 5 programs compiled for use with CID. These features include commands with a COBOL-like syntax and the capability of symbolically referencing locations within the program. The commands available only for programs compiled for use with CID are indicated in appendix D.

You can compile a COBOL 5 program for use with CID by entering the COBOL5 control statement while debug mode is turned on. For purposes of this user's guide, it is assumed that COBOL programs to be executed under CID control are compiled for use with CID; therefore, in the discussions of the CID capabilities, no distinction is made between standard CID features and the special features available to COBOL programs. It is possible, though more difficult, to use CID with programs not compiled for use with CID. Refer to the CYBER Interactive Debug reference manual for a description of this capability.

PROGRAMMING FOR EASE OF DEBUGGING

Even though CID offers many useful debugging features, a well-written program is much easier to debug than a badly-written program. You can use CID more effectively if you follow good programming practices such as:

Using an accepted design methodology

Using rules of structured programming style

CID should not be considered a substitute for good programming practices.

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

If the special COBOL features are to be used in a debug session, the program must be compiled for use with CID. Additional code is generated when the program is compiled for use with CID; this additional code increases execution time slightly.

The effect of CID on the field length is described as follows. CID consists of several parts that are similar to overlays. The main part is always in memory and is approximately 4000_8 words long. The other parts are exchanged in memory with the program being debugged and can require up to 54000_8 words of memory. Therefore, if your program is smaller than 54000_8 words, the field length requirement when your program is debugged under CID is approximately 60000_8 words. If your program is larger than 54000_8 words, the field length requirement is 4000_8 words larger than the size of your program.

RESTRICTIONS ON PROGRAMS THAT CAN BE DEBUGGED USING CID

CID cannot be used to debug programs that:

Have dynamically loaded subprograms.

Have fixed overlayable or independent segments.

Use the Message Control System (MCS).

Use the CYBER Database Control System (CDCS).

BATCH MODE DEBUGGING

Although CID is intended to be used interactively, it can be used in batch mode. Batch mode debugging is described in appendix C.

GETTING STARTED 2

This section summarizes the operations necessary to conduct a debug session and introduces several CYBER Interactive Debug (CID) notation conventions. At the end of the section, several basic commands are presented and used in a sample session. These commands enable you to conduct a simple but useful debug session.

BEGINNING A DEBUG SESSION

To execute a program under CID control (and to make use of the COBOL capabilities), you must compile and execute the program with debug mode turned on. Debug mode is turned on by a system control statement.

DEBUG CONTROL STATEMENT

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

```
    DEBUG
or
    DEBUG(ON)
```

When a COBOL program is compiled in debug mode, the program is compiled for use with CID; special symbol tables used by CID are generated as part of the object code. When the program is subsequently executed in debug mode, all of the CID features can be used. Note that a program that has not been compiled for use with CID can still be executed in debug mode, but many of the features described in this guide will not be available.

When debug mode is on, you can interact with the operating system and perform all other terminal activities in a normal manner; only compilations and relocatable loads are affected.

The statement to deactivate debug mode is:

```
    DEBUG(OFF)
```

When debug mode is off, programs that were compiled for use with CID execute normally (although less efficiently than programs that were not compiled for use with CID). It is necessary to enter DEBUG(OFF) only if you do not wish subsequent compilations or executions to occur under CID control.

EXECUTING UNDER CID CONTROL

A debug session consists of the sequence of interactions between you and CID which take 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.

The debug session is initiated by entering the name of the file that contains your binary program after the compilation has completed. (Usually, this file is LGO.) The system loads the CID program module, 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. In this figure and all terminal sessions in this guide, user input is lowercase, and system response is uppercase.

```
NOS:
   /debug
   DEBUG.
   /cobol5,i=program
       .
       .
       .
   /lgo
    CYBER INTERACTIVE DEBUG
   ?


NOS/BE:
   COMMAND- debug
   COMMAND- cobol5,i=program
       .
       .
       .
   COMMAND- lgo
   CYBER INTERACTIVE DEBUG
   ?
```

Figure 2-1. Initiating a Debug Session

Debugging a program can require more than one debug session. If this is the case, you can terminate the current session and initiate a new session. Note that once a program has been compiled in debug mode, it is not necessary to recompile in order to conduct another debug session with the same program. You can initiate another session merely by entering the binary file name (the normal method of executing a program).

ENTERING CID COMMANDS

The CID prompt for your response is a question mark (?). In response to the ? character, enter a CID command and press the transmission key (RETURN on most terminals). CID then processes the command, issuing a message if appropriate, and issues another ? prompt. CID continues to issue prompts after processing commands until you enter the command to resume execution of your program, or until you terminate the session.

If you enter a command incorrectly, CID displays a diagnostic message. One such message is:

```
        *ERROR - UNKNOWN COMMAND
```

If this message appears, determine the correct format, and reenter the command. You can use the HELP command, described later in this section, for assistance with command formats.

SHORT AND LONG FORM OF CID COMMANDS

Many CID commands have a long form that spells out the name of the command and a short form that abbreviates the command name and parameters. For example, the long-form command

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

can be expressed as

```
    ST,L,*
```

In this guide, both forms are described for commands of this sort. However, to make sample debug sessions as understandable as possible, long forms are usually shown. (The D command described in section 3 is always shown in short form to avoid confusion with the COBOL CID DISPLAY command described in this section.) You are encouraged to use the short forms as you become familiar with CID; they have the same effects as long forms.

A more detailed explanation of CID command syntax and a list of long and short forms are given in appendix D.

MULTIPLE COMMAND LINES

You can enter several CID commands on the same line if you separate them with semicolons (;). For example, entering

```
    DISPLAY A; GO
```

has the same effect as entering

```
    DISPLAY A
    GO
```

REFERENCING SOURCE STATEMENTS

Many of the CID command formats require you to indicate a specific statement within the program you are debugging. Source statements are referenced either by line sequence number or procedure name using the notations described in the following paragraphs.

LINE NUMBER SPECIFICATION

The notation for specifying a line number is:

```
    L.n
```

where n is the statement sequence number or the number indicated on the compiler-generated source listing for programs without sequence numbers. This notation denotes the source line having the specified sequence number. Leading zeros can be omitted. Some examples of line number references are as follows:

L.130	This example refers to the beginning of the line with sequence number 130.
L.26	This example refers to the beginning of the line with sequence number 26.

Some lines cannot be referenced through line number specification. Only procedure-name lines and lines in the Procedure Division that contain COBOL verbs can be referenced.

Line-number references refer to the statement with the first verb on the line. The beginning of the statement is referenced.

PROCEDURE-NAME SPECIFICATION

The notation for specifying the beginning of a paragraph or section in the Procedure Division is:

```
    PR.procedure-name
```

The procedure-name can take one of these forms:

```
    paragraph-name

    section-name

    paragraph-name OF section-name
```

The procedure-name specification refers to the line that contains the procedure-name (not to the entire procedure). When specifying a paragraph-name that is used in more than one section in the program, you must qualify the paragraph-name with a section-name to ensure that CID locates the correct paragraph.

Some examples of procedure-name references are as follows:

```
PR.GET-VALUES
```

This example refers to the line identifying the GET-VALUES paragraph or section.

```
PR.GET-VALUES OF READ-IN-DATA
```

This example refers to the line identifying the GET-VALUES paragraph in the READ-IN-DATA section.

SOME ESSENTIAL COMMANDS

The following paragraphs describe several CID commands that enable you to conduct simple debug sessions. These are the GO command, the QUIT command, the DISPLAY command, the SET,BREAKPOINT command, and the HELP command. (All but the HELP command are described in greater detail in later sections.) The command forms presented here allow you to debug programs consisting of a single program unit only. To debug programs containing multiple program units (main program and subprograms), you must be familiar with concepts described in section 4.

GO COMMAND

The command to initiate or resume program execution is:

```
GO
```

If entered at the beginning of the debug session, this command initiates program execution. If entered after execution has been suspended, this command causes execution to resume at the statement where it was suspended.

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

QUIT COMMAND

The command to terminate a debug session is:

```
QUIT
```

In response to the QUIT command, CID displays the following message:

```
DEBUG TERMINATED
```

The QUIT command causes an exit from the current debug session and a return to system command mode. Files accessed by the COBOL program are closed. Note, however, that debug mode remains on until DEBUG(OFF) is specified. You can initiate another debug session for the same program, without recompiling, by entering the binary file name (as described under Beginning a Debug Session).

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. You can terminate a debug session any time you have control (CID has issued a ? prompt). The object program can then be executed normally, or it can be executed again under CID control.

DISPLAY COMMAND

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

```
DISPLAY list
```

where list is a list of identifiers and literals. This command has the same format as the COBOL DISPLAY statement. Identifiers can contain subscripts and reference modifiers. This command lists the values of the specified data-items.

Examples of the DISPLAY command are as follows:

```
DISPLAY ACCOUNT-NUMBER
```

This command displays the value of the data-item ACCOUNT-NUMBER.

```
DISPLAY ACCT(1), "  ", NAME OF CUST (1:5)
```

This command displays the value of the first element of the table ACCT, two spaces, and the first five characters in the data-item NAME in group-item CUST.

SET,BREAKPOINT COMMAND

A breakpoint is a location within a program where execution is to be suspended. The command to establish a breakpoint has the form

```
SET,BREAKPOINT,loc
```

where loc is a line number specification (L.n) or procedure-name specification (PR.procedure-name) as described under Referencing Source Statements. The short form of SET,BREAKPOINT is SB. When the specified statement is reached in the flow of execution, control transfers to CID which then allows you to enter CID commands. Typically, commands are entered to examine the values of program variables, and execution is resumed.

Examples of the SET,BREAKPOINT command are as follows:

```
SET,BREAKPOINT,L.114
```

This command sets a breakpoint at line 114.

```
SET BREAKPOINT,PR.GET-VALUES IN READ-DATA
```

This command sets a breakpoint at the line containing the paragraph-name GET-VALUES in the READ-DATA section.

```
SB,PR.GET-VALUES IN READ-DATA
```

This command has the same effect as the previous example.

You can establish breakpoints at any time in the debug session when execution is suspended and CID has issued a ? prompt.

A breakpoint can be established at any line in the Procedure Division that contains a COBOL verb. Execution is suspended before the statement containing the first verb is executed. Only one breakpoint can be set at a particular line.

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 location (L.n or PR.procedure-name) where the breakpoint was set. Up to 16 breakpoints can be in effect at a given time; each breakpoint is assigned a number in the range 1 through 16.

HELP COMMAND

CID provides a HELP command that displays a brief summary of information about specific CID subjects and commands. You can enter 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 useful form of the HELP 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
```

where command is any CID 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 2-2, which shows the entry of the command HELP,SET, BREAKPOINT to display a summary of the command parameters.

SUMMARY

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 useful features of CID.

To conduct a simple debug session using the information provided in this section, you can follow these steps:

1. Type DEBUG to turn on debug mode.

2. Compile and load your program in a normal manner. Control transfers to CID when execution begins. CID displays a message at the terminal and waits for your input.

3. Set breakpoints as desired. To set a breakpoint at a line number, enter

   ```
   SET,BREAKPOINT,L.n
   ```

   where n is the line number. To set a breakpoint at the beginning of a paragraph or section in the Procedure Division, enter

   ```
   SET,BREAKPOINT,PR.procedure-name
   ```

   where procedure-name is the name of the paragraph or section.

```
? 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 <LOCATION> HAS BEEN HIT <FIRST> TIMES. BUT, IT WILL BE
 HONORED WHEN <LOCATION> IS HIT THE <FIRST>TH TIME AND EACH
 <STEP>TH TIME AFTER THAT AS LONG AS <LAST> IS NOT 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 2-2. Example of HELP Command

4. Enter GO to begin execution of your program.

   CID executes your program in a normal manner, but returns control to you when a breakpoint occurs or the program terminates.

5. At this point, you can display the values of program variables with the statement:

       DISPLAY list

   To resume execution, enter GO.

6. Enter QUIT to terminate the session. Enter DEBUG(OFF) to turn off debug mode.

Debug sessions can become complicated. Always try to keep debug sessions short and simple. If necessary, correct known bugs, recompile your program, and conduct additional debug sessions.

SAMPLE DEBUG SESSION

The commands described in this section are used to conduct a simple debug session. As you study the examples in this guide, keep in mind that these examples are intended to illustrate the various CID features; they are not intended to present a suggested sequence of commands for debugging all programs. The actual commands that you enter in a debug session depend on your program and, often, on your intuition.

The input data shown in figure 2-3 is a list of bids submitted for one item at an auction. The COBOL program in figure 2-4 is intended to sort the bids in descending order to facilitate finding the highest bid. However, the program contains an error: The bids are never sorted.

You might try to debug the program under CID control, as shown in figure 2-5. In this example, each bid is displayed as it is read to see that it has the correct value. Then the sorting input procedure is monitored; this procedure appears to execute correctly. Next, the sorting output procedure is monitored, and it is seen that data is not moved from the sort file back to the table of bids. Analysis of the program shows that a period left off of the RETURN statement in the sorting output procedure caused a MOVE statement to be part of the AT END clause. One error has been found.

At this point, the program should be corrected and recompiled. The program can then be reexecuted to see if other errors are present.

Figure 2-6 shows the same debug session using short forms of the CID commands.

```
444332
011023
648234
003325
```

Figure 2-3. Input File BIDS

```
 1          IDENTIFICATION DIVISION.
 2          PROGRAM-ID.  SORT-BIDS.
 3          *
 4          *    THIS PROGRAM SORTS A LIST OF BIDS SUBMITTED FOR ONE ITEM
 5          *    AT AN AUCTION.  EACH INPUT LINE TAKES THE FORM:
 6          *      BID                      PICTURE 9999V99.
 7          ENVIRONMENT DIVISION.
 8          CONFIGURATION SECTION.
 9          SOURCE-COMPUTER.  CYBER-170.
10          OBJECT-COMPUTER.  CYBER-170.
11          INPUT-OUTPUT SECTION.
12          FILE-CONTROL.
13              SELECT IN-FILE ASSIGN TO "BIDS".
14              SELECT OUT-FILE ASSIGN TO "OUTPUT".
15              SELECT SORT-FILE ASSIGN TO SFILE.
16          DATA DIVISION.
17          FILE SECTION.
18          FD IN-FILE
19              LABEL RECORD IS OMITTED
20              DATA RECORD IS LINE-IN.
21          01  LINE-IN.
22              05  BID                     PICTURE 9999V99.
23              05  FILLER                  PICTURE X(4).
24          FD OUT-FILE
25              LABEL RECORD IS OMITTED
26              DATA RECORD IS LINE-OUT.
27          01  LINE-OUT.
28              05  FILLER                  PICTURE X(10).
29              05  BID                     PICTURE $9999.99.
30          SD  SORT-FILE
31              RECORD CONTAINS 6 CHARACTERS
32              DATA RECORD IS SORT-RECORD.
33          01  SORT-RECORD.
34              05  BID                     PICTURE 9999V99
```

Figure 2-4. COBOL Program (Sheet 1 of 2)

```
WORKING-STORAGE SECTION.
01  BID-INFORMATION.
    05  NUMBER-OF-BIDS          PICTURE 99V.
    05  BID-TABLE               OCCURS 10 TIMES
                                INDEXED BY BID-INDEX.
        10  BID                 PICTURE 9999V99.
PROCEDURE DIVISION.
INITIALIZATION SECTION.

OPEN-FILES.
    OPEN INPUT IN-FILE
         OUTPUT OUT-FILE.
INITIALIZE-VALUES.
    MOVE ZERO TO NUMBER-OF-BIDS.
PROCESS-A-BID SECTION.
READ-BIDS.
    READ IN-FILE AT END GO TO SORTING.
    ADD 1 TO NUMBER-OF-BIDS.
    MOVE  LINE-IN TO BID OF BID-TABLE (NUMBER-OF-BIDS).
    GO TO READ-BIDS.
SORTING SECTION.
SORT-THE-BIDS.
    SORT SORT-FILE
            ON DESCENDING KEY BID OF SORT-RECORD
          INPUT PROCEDURE IS SORT-IN-PROC
          OUTPUT PROCEDURE IS SORT-OUT-PROC.
    GO TO WRITE-RESULTS.
SORT-IN-PROC SECTION.
START-OF-SECTION.
    PERFORM VARYING BID-INDEX FROM 1 BY 1
        UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS
    RELEASE SORT-RECORD FROM BID OF BID-TABLE (BID-INDEX)
    END-PERFORM.
SORT-OUT-PROC SECTION.
START-OF-SECTION.
    PERFORM SORTING-PARAGRAPH VARYING BID-INDEX FROM 1 BY 1
        UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
    GO TO END-OF-SECTION.
SORTING-PARAGRAPH.
      RETURN SORT-FILE RECORD
          AT END GO TO END-OF-SECTION
      MOVE SORT-RECORD TO BID OF BID-TABLE (BID-INDEX).
END-OF-SECTION.
WRITE-RESULTS SECTION.
WRITE-BIDS.
    PERFORM WRITE-ONE-BID VARYING BID-INDEX FROM 1 BY 1
        UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
    GO TO END-OF-RUN
WRITE-ONE-BID.
    MOVE BID OF BID-TABLE (BID-INDEX) TO BID OF LINE-OUT.
    WRITE LINE-OUT.
END-OF-RUN SECTION.
CLOSE-FILES.
    CLOSE IN-FILE, OUT-FILE.
STOP-RUN.
    STOP RUN.
```

Figure 2-4. COBOL Program (Sheet 2 of 2)

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,pr.sorting           Set a breakpoint at the beginning of the SORTING section.
? go                                  Initiate execution.
 *B #1, AT PR.SORTING                 CID suspends execution when the breakpoint location is reached.
? display number-of-bids
 4
? display bid of bid-table (1)
 4443.32
? display bid of bid-table (2)
 110.23                               Display the bids to see that they were read correctly.
? display bid of bid-table (3)
 6482.34
? display bid of bid-table (4)
 33.25
? set,breakpoint,pr.sort-out-proc     Set a breakpoint at the beginning of the SORT-OUT-PROC paragraph.
? set,breakpoint,l.66                 Set a breakpoint at line 66 to monitor the sorting input procedure.
? go                                  Resume execution.
 *B #3, AT L.66
? display sort-record                 The sort-file record is empty, because the breakpoint occurs before line 66 is executed.

? go
 *B #3, AT L.66
? display sort-record
 444332
? go
 *B #3, AT L.66
? display sort-record
 011023                               Monitoring of the sorting input procedure shows that, most likely, the procedure executes
? go                                  correctly.
 *B #3, AT L.66
? display sort-record
 648234
? go
 *B #2, AT PR.SORT-OUT-PROC
? display sort-record
 003325
? set,breakpoint,l.73                 Set a breakpoint at line 73 to monitor the sorting output procedure.
? go
 *B #4, AT L.73
? go
 *B #4, AT L.73
? display bid of bid-table (1)        The first bid moved to BID-TABLE should be the highest, that is 6482.34.
 4443.32
? set,breakpoint,l.76                 Set a breakpoint at line 76 to see if the MOVE statement is executed.
? go
 *B #4, AT L.73
? go                                  The breakpoint at line 76 does not occur; the MOVE statement is never executed.  Analysis of
 *B #4, AT L.73                       the program shows that the period was left off of the end of the AT END clause of the RETURN
? quit                                statement in line 75.  The MOVE statement is considered part of the AT END clause.
FILE OUTPUT NOT CLOSED AT STOP RUN OR CANCEL - C
                                      Terminate the debug session.  The DEBUG(OFF) control statement must be entered if the next
                                      compilation or execution will not use CID.
```

Figure 2-5. Sample Debug Session

```
 CYBER INTERACTIVE DEBUG
? sb,pr.sorting
? go
 *B #1, AT PR.SORTING
? display number-of-bids
  4
? display bid of bid-table (1)
  4443.32
? display bid of bid-table (2)
  110.23
? display bid of bid-table (3)
  6482.34
? display bid of bid-table (4)
  33.25
? sb,pr.sort-out-proc
? sb,l.66
? go
 *B #3, AT L.66
? display sort-record

? go
 *B #3, AT L.66
? display sort-record
 444332
? go
 *B #3, AT L.66
? display sort-record
 011023
? go
 *B #3, AT L.66
? display sort-record
 648234
? go
 *B #2, AT PR.SORT-OUT-PROC
? display sort-record
 003325
? sb,l.73
? go
 *B #4, AT L.73
? go
 *B #4, AT L.73
? display bid of bid-table (1)
  4443.32
? sb,l.76
? go
 *B #4, AT L.73
? go
 *B #4, AT L.73
? quit
FILE OUTPUT NOT CLOSED AT STOP RUN OR CANCEL - C
```

Figure 2-6. Same Session Using Short Forms

ADVANCED DEBUGGING TECHNIQUES 3

The preceding section presented some elementary commands that can be used to conduct a simple debug session. This section provides you with additional information on the commands presented in section 2 and describes some other commands and CID features that allow you to make more productive use of CID. The commands discussed in this section enable you to:

Suspend program execution

Display current program values at the terminal while execution is suspended

Change current program values while execution is suspended

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, then reenter the command. Figure 3-1 illustrates some typical error messages. The first message is caused by a misspelled DISPLAY command. In the second example, the command is syntactically correct but the program does not contain an IN-PROC paragraph; correcting the paragraph-name to SORT-IN-PROC makes the SET,BREAKPOINT command acceptable.

```
? disply number-of-bids
 *ERROR - UNKNOWN COMMAND
? display number-of-bids
  4
? set,breakpoint,pr.in-proc
 *ERROR - NO PROCEDURE NAME IN-PROC
? set,breakpoint,pr.sort-in-proc
?
```

Figure 3-1. 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 - text
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.

Some examples of warning messages are illustrated in figure 3-2. The first message is generated when an attempt is made to set a breakpoint in line 5, outside the Procedure Division. In this case, line 50 was intended. The correct command is entered in response to the OK? prompt. The second message

```
? set,breakpoint,l.5
 *WARN - LINE 5 NOT EXECUTABLE - LINE 41 WILL BE USED
 OK ? set,breakpoint,l.50
? clear,breakpoint
 *WARN - ALL WILL BE CLEARED
 OK ? ok
?
```

Figure 3-2. Debug Session Illustrating Warning Messages

occurs after a CLEAR,BREAKPOINT command is entered. CID warns that this command removes all existing breakpoints, and allows you to reconsider. An affirmative response is entered, and CID executes the CLEAR,BREAKPOINT command.

Warning messages can be suppressed by an option on the SET,OUTPUT command, described later in this section under Control of CID Output. 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.

BREAKPOINTS AND TRAPS

When conducting a debug session, you must initially provide for gaining interactive control at some point within your program. CID provides breakpoints and traps for this purpose.

A breakpoint (introduced in section 2) causes program execution to be suspended when a specified statement is reached in the flow of execution. A trap causes execution to be suspended when a specified condition is detected during execution. Both breakpoints and traps cause CID to give control to you so that you can examine and alter the status of your program at various points during execution.

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

In most debugging situations, breakpoints, rather than traps, are recommended for suspending execution. Traps can be useful in certain cases, but some trap types require you to be familiar with compiled object code; only trap types useful to most COBOL programmers are covered here. Breakpoints allow you to suspend execution at any executable statement in your program and can, in most cases, be substituted for traps.

Breakpoints and traps exist only for the duration of a debug session. Once a session is terminated, all breakpoints and traps set during a session cease to exist (unless they are saved on a file as described in section 5). An object program is not permanently altered by any breakpoints or traps established during a session.

CID provides commands that enable you to:

Establish breakpoints and traps

Display a list of existing breakpoints and traps

Remove existing breakpoints and traps

Save breakpoint and trap definitions on a separate file for use in a later debug session

SUSPENDING EXECUTION WITH BREAKPOINTS

A breakpoint is a mechanism established at a specified location within a program such that when the location is reached during program execution, control passes to CID which displays a message and gives control to you.

The SET,BREAKPOINT command (described in section 2) can be used to set breakpoints at the beginning of lines, paragraphs, or sections in the Procedure Division.

It is important to note that breakpoints set at lines suspend execution before the line is executed. For example, assume a program contains the following statements

```
31      MOVE ZERO TO COUNT.
32      ADD 1 TO COUNT.
```

and that a breakpoint is set at line 32. Then when line 32 is reached, execution is immediately suspended before the statement at line 32 is executed. Thus, COUNT has the value zero, not one. When execution is resumed, the statement at line 32 is executed and the value of COUNT is increased to one. Breakpoints set at paragraphs or sections in the Procedure Division are set at the line containing the procedure-name.

FREQUENCY PARAMETERS

When a breakpoint is set, execution is suspended each time the breakpoint location is reached. For example, if a breakpoint is set within a paragraph specified in a PERFORM statement, suspension occurs on each pass through the paragraph. This can result in many unnecessary suspensions during the course of a debug session. To alleviate this situation, CID provides frequency parameters for the SET,BREAKPOINT command that are extremely useful for debugging sections of a program which are executed frequently. The command appears as follows

```
SET,BREAKPOINT,loc,first,last,step
```

where first, last, and step are frequency parameters. The parameter first indicates the first time the breakpoint suspends execution. The parameter last indicates the last time the breakpoint suspends execution. The parameter step indicates how often the breakpoint suspends execution. For example, the command

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

sets a breakpoint at line 50 which suspends execution the tenth time the statement is reached and every fifth time thereafter, up through the hundredth time.

As an example of the use of the frequency parameters, consider the statements shown in figure 3-3. To examine the progress of the COMPUTE statement, you can set a breakpoint at paragraph PAR1, specifying frequency parameters to suspend execution at an interval rather than on each pass through the loop. For example

```
    SET,BREAKPOINT,PR.PAR1,2,1000,100
```

sets a breakpoint that suspends execution on every hundredth pass through the paragraph, starting with the second pass.

```
          PERFORM PAR1
            VARYING I FROM 1 BY 1
            UNTIL I IS GREATER THAN 1000.
                  .
                  .
                  .
      PAR1.
          COMPUTE RATE(I) = 1.57 * B(I).
```

Figure 3-3. Frequency Parameter Example

LISTING BREAKPOINTS

You can display a list of breakpoints defined in a debug session by entering the LIST,BREAKPOINT command:

```
    LIST,BREAKPOINT,*
```

This command displays a list of all breakpoints in the program. The short form of LIST,BREAKPOINT is LB.

The LIST,BREAKPOINT command lists the breakpoints that exist at the time the command is entered. The list contains the number and location of each breakpoint in the following form

```
    *B #i = loc
```

where i is the breakpoint number assigned by CID and loc is the location (line number or procedure name) where the breakpoint is set. If frequency parameters were specified when the breakpoint was set, they also appear in the list.

You can list a specific breakpoint by its breakpoint number by entering the command

```
    LIST,BREAKPOINT,#n
```

where n is the number of the breakpoint.

If no breakpoints exist when a LIST,BREAKPOINT command is entered, CID displays the following message:

```
    NO BREAKPOINTS
```

Examples of the LIST,BREAKPOINT command are as follows:

```
    LIST,BREAKPOINT,#4
```

This command lists breakpoint 4.

```
    LIST,BREAKPOINT,L.137
```

This command lists the breakpoint at line 137.

```
    LB,L.137
```

This command has the same effect as the previous example.

REMOVING BREAKPOINTS

As the debug session proceeds, breakpoints set early in the session might no longer be desired. You can remove breakpoints during a debug session by entering one of the following commands:

```
    CLEAR,BREAKPOINT,*
```

This command clears all currently defined breakpoints.

```
    CLEAR,BREAKPOINT,loc-list
```

This command clears the breakpoints from the specified locations; loc-list is a list of locations separated by commas. Each location has one of the following forms:

```
    L.n
```

This form refers to line n.

```
    Pr.procedure-name
```

This form refers to the beginning of a paragraph or section in the Procedure Division.

```
    #n
```

This form refers to breakpoint number n.

If a breakpoint does not exist at a specified location, CID displays the message

```
    NO BREAKPOINT loc
```

where loc is the breakpoint location, and no action is taken. The short form of CEAR,BREAKPOINT is CB.

Examples of the CLEAR,BREAKPOINT command are as follows:

```
    CLEAR,BREAKPOINT,L.114,L.220,PR.SORTING
```

This command removes the breakpoints from lines 114 and 220 and from the line identifying the paragraph or section named SORTING.

```
    CLEAR,BREAKPOINT,#3,#5,#6
```

This command removes breakpoints 3, 5, and 6.

```
    CB,#3,#5,#6
```

This command has the same effect as the previous example.

SUSPENDING EXECUTION WITH TRAPS

Traps suspend execution and give you control whenever specified conditions occur. For example, traps can give you control when you enter a terminal interrupt, when execution terminates, or when the beginning of a new line is reached.

TRAP USAGE

The most useful traps to the COBOL programmer are the LINE and PROCEDURE traps. (The END, ABORT, and INTERRUPT traps are also used, but they are established automatically by CID.) The remaining CID traps are oriented toward COMPASS programs and are not described in this guide. The traps described in this section are listed in table 3-1. See the CYBER Interactive Debug reference manual for information on other types of traps.

When a trap condition is detected, execution is suspended, CID gains control, and CID issues a message identifying the trap, followed by a ? prompt for your input. The message gives information about the trap, including the trap type, the trap number, and the location (L.n or PR.procedure-name) where the trap occurred. The trap number is a decimal integer assigned by CID. An example of a trap message is:

```
*T #3, LINE AT L.345
?
```

In this example, a LINE trap has been detected at line 345; this trap was the third one established.

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 the GO command.

Traps suspend execution when a specific event occurs. Some traps suspend execution before the event, while others suspend execution after the event. This is an important distinction because it can affect the status of variables you are displaying or altering. For example, assume that execution is suspended at line 32 of the following program segment:

```
31      MOVE ZERO TO COUNT.
32      ADD 1 TO COUNT.
```

If the trap suspended execution before the statement at line 32 was executed, COUNT contains zero. If the trap suspended execution after the statement was executed, COUNT contains one. Table 3-1 indicates, for each trap, the point in execution where CID gets control.

The traps described in this section are of two types: default traps and user-established traps. User-established traps are set by the SET,TRAP command. The default traps always exist; it is not necessary to specify a SET,TRAP command for these traps. Table 3-1 indicates default traps and user-established traps.

DEFAULT TRAPS

CID provides default traps that are automatically set 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.

Together, the END and ABORT traps 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.

TABLE 3-1. TRAP TYPES

Trap Type	Short Form	Condition	Established by	User Gets Control
LINE	L	Beginning of an executable source line that is not continued from a previous line. The LINE trap also occurs at the beginning of procedure-name lines.	User	Before the line is executed
PROCEDURE	PROC	Procedure-name line.	User	Before the line is executed
INTERRUPT		User interrupt.	Default	After the interrupt
END		Normal program termination (through a STOP RUN statement).	Default	After termination occurs
ABORT		Abnormal program termination.	Default	After termination occurs

END Trap

The END trap gives control to CID 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.

Note that the debug session does not end when your program terminates. The END trap allows you to enter commands and continue the session until you enter the QUIT command.

When the program terminates, CID gains control and issues the message:

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

CID permanently assigns the number 17 to the END trap. The line where execution terminated is given by n. 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 enter a GO command following an END trap.

ABORT Trap

The ABORT trap is useful because it allows you to gain control when abnormal termination of program execution occurs. Program values can be examined as they exist at the precise time of termination.

To illustrate how the ABORT trap works, a program containing a reference modification error is executed under CID control. The source listing and debug session are shown in figure 3-4. The MOVE statement in line 13 specifies an out-of-range reference modification. As shown in the figure, you can observe program values after the ABORT trap occurs.

The ABORT trap can also occur when the execution time limit is exceeded. On NOS/BE, the trap occurs immediately after the time limit is exceeded.

On NOS, the operating system first gains control. You can then direct the operating system to continue or to stop execution. If you direct the operating system to continue execution, the program resumes execution and the ABORT trap does not occur. However, if you direct the operating system to stop execution, CID gains control and the ABORT trap occurs.

Deciding whether or not to continue execution depends on the reason the time limit was exceeded. If you program is executing an infinite loop, you want execution to stop. However, if your program simply requires more time to execute, you want execution to continue. If you are not sure about whether to continue or stop execution, it is usually best to stop execution and consult your program listing to see if your program has an infinite loop.

On both operating systems, you are given a small amount of time to execute CID commands; if this time is exceeded, the debug session is terminated.

```
1            IDENTIFICATION DIVISION.
2            PROGRAM-ID. SHOW-ABORT-TRAP.
3            ENVIRONMENT DIVISION.
4            DATA DIVISION.
5            WORKING-STORAGE SECTION.
6            01  A                          PICTURE X(10)
7                                           VALUE "ABCDEFGHIJ".
8            01  B                          PICTURE 9V.
9            01  C                          PICTURE X(5).
10           PROCEDURE DIVISION.
11           BAD-REFERENCE-MODIFICATION.
12               MOVE 8 TO B.
13               MOVE A (B : 5) TO C.
14               STOP RUN.
```

```
 CYBER INTERACTIVE DEBUG
? go
 *T #18, ABORT ILLEGAL REFERENCE MODIFICATION IN L.13 <------ ABORT trap suspends execution in line 13.
? display a (b : 5)             )
 *ERROR - REFERENCE MOD OUT OF RANGE |
? display a, b                   }  <------------------------- Program values are displayed while execu-
 ABCDEFGHIJ 8                    |                             tion is suspended.
? display a (b : 2)              |
 HI                             )
? quit
 DEBUG TERMINATED
```

Figure 3-4. Program and Debug Session Illustrating ABORT Trap

When the ABORT trap occurs, the number of the line in which execution stopped is displayed in the trap report message. You can then analyze your program listing to find the cause of abnormal termination. For example, in the case of a time limit ABORT trap, you could look for an infinite loop in the area where the time limit occurred. Sometimes it is useful to initiate another debug session and set breakpoints to monitor program values before the time limit is reached.

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

INTERRUPT Trap

The INTERRUPT trap gives control to CID when you issue a terminal interrupt. The procedure for issuing a terminal interrupt depends on the terminal type and on the interactive communication system in use. See interrupt in the Glossary, appendix C.

When you enter the appropriate interrupt sequence, the process currently active is interrupted; CID gets control, issues the INTERRUPT trap message, and gives control to you. The INTERRUPT trap can be used to terminate excessive output to the terminal, although 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.

The INTERRUPT trap is permanently assigned the number 19 by CID.

USER-ESTABLISHED TRAPS

In addition to the default traps, CID provides traps that can be established whenever you have control. Up to 16 user-established traps can be in effect at a given time.

SET,TRAP Command

The traps described in the following paragraphs are established with the SET,TRAP command. This command has the form

```
SET,TRAP,type,scope
```

where type is one of the trap types listed in table 3-1, and scope is one of the notation forms listed in table 3-2. The short form of SET,TRAP is ST.

TABLE 3-2. TRAP SCOPE PARAMETER

Scope	Trap is Set
*	Everywhere
L.n	In line n
L.n...L.m	Everywhere within the range of lines n through m (n<m)

The scope parameter of the SET,TRAP command specifies the program locations for which the trap is effective. The scope of a trap can be a single COBOL line, several lines, or the entire program. To specify a single line, you enter L.n as the scope of the trap. To specify the entire program, you enter an asterisk (*) as the scope.

To specify several lines as the trap scope, you specify the following ellipsis notation

```
L.n...L.m
```

where n is less than m. This notation refers to all of the lines in the source program consisting of lines n through m. You must not enter spaces between the three periods in this notation.

Not all forms of the scope listed in table 3-2 are valid for all CID trap types; valid forms depend on the particular type of trap you set. The forms listed in table 3-2 are valid for the particular trap types described in this user's guide.

Traps can be established whenever program 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.

LINE Trap

The LINE trap suspends program execution and gives control to CID immediately prior to execution of each executable COBOL line that is within the scope and that is not continued from a previous line. This trap allows you to examine and alter program values before each statement is executed. The command to set a LINE trap has the form

```
SET,TRAP,LINE,scope
```

where scope has one of the following forms:

`*`

The trap is set for each line in the entire program.

`L.n...L.m`

The trap is set for each of lines n through m (n < m).

Examples of setting a LINE trap are as follows:

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

This command suspends execution before each executable line in the entire program.

```
SET,TRAP,LINE,L.221...L.254
```

This command suspends execution before each of lines 221 through 254 is executed.

```
ST,LINE,L.221...L.254
```

This command has the same effect as the previous example.

PROCEDURE Trap

The PROCEDURE trap suspends program execution and gives control to CID immediately prior to execution of any procedure-name line at the beginning of a paragraph or section in the Procedure Division. The command to set a PROCEDURE trap has the form:

```
SET,TRAP,PROCEDURE,scope
```

The scope has one of the following forms:

```
*
```

The trap is set for each procedure-name line in the entire program.

```
L.n...L.m
```

The trap is set for each procedure-name line in lines n through m (n < m).

Examples of setting a PROCEDURE trap are as follows:

```
SET,TRAP,PROCEDURE,*
```

This command suspends execution at each procedure-name line in the entire program.

```
SET,TRAP,PROCEDURE,L.34...L.93
```

This command suspends execution at each procedure name line in lines 34 through 93.

```
ST,PROCEDURE,L.34...L.93
```

This command has the same effect as the previous example.

LISTING TRAPS

To display a list of traps defined for a debug session, enter the command:

```
LIST,TRAP,*
```

This command displays the number, type, and scope of all traps that exist at the time the command is entered. The short form of LIST,TRAP is LT. LIST,TRAP output has the following form:

```
T #n = type scope
```

where n is the trap number assigned by CID, type is the trap type as listed in table 3-1, and scope is the trap scope in the form that you specified in the SET,TRAP command.

You can list a specific trap by its trap number by entering the command

```
LIST,TRAP,#n
```

where n is the number of the trap.

If no traps exist when LIST,TRAP is entered, CID displays the message:

```
NO TRAPS
```

Examples of the LIST,TRAP command are:

```
LIST,TRAP,#7
```

This command lists trap 7.

```
LT,#7
```

This command has the same effect as the previous example.

REMOVING TRAPS

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

```
CLEAR,TRAP,*
```

This command removes all of the traps that you have defined.

```
CLEAR,TRAP,type,*
```

This command removes all traps of the specified type.

```
CLEAR,TRAP,number-list
```

This command removes the traps identified by the specified number-list; number-list is a list of trap numbers of the form #n separated by commas.

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 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 described in section 5. The following are examples of the CLEAR,TRAP command:

```
CLEAR,TRAP,LINE,*
```

This command clears all LINE traps.

```
CLEAR,TRAP,#2,#4,#5
```

This command clears the traps identified by trap numbers 2, 4, and 5.

SUMMARY OF BREAKPOINT AND TRAP CHARACTERISTICS

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

You can set, clear, or list breakpoints and traps any time CID has control and has prompted you for 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.

Breakpoints and traps exist for the duration of the debug session unless removed by the CLEAR command.

The frequency parameters of the SET,BREAKPOINT command can be used to avoid suspending execution every time a location is reached.

CID automatically establishes END, ABORT, and INTERRUPT traps so that you receive control on program termination or terminal interrupt, even if you have not explicitly established any breakpoints or traps.

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 breakpoint or trap location is executed in the normal manner when execution is resumed.

EXECUTING A FEW LINES AT A TIME

The STEP command can be used to execute a few lines and then give you control. The format of the STEP command is as follows:

```
    STEP,n,LINE
```

The parameter n indicates the number of lines to be executed. The short form of STEP is S.

When you enter the STEP command, CID resumes or initiates program execution and counts each line when the first COBOL verb in the line is reached (or at the beginning of the line in the case of procedure-name lines). CID counts only procedure-name lines and lines that contain verbs. When the number of lines counted is equal to n, CID suspends execution and displays the following message:

```
    *S LINE AT L.n
```

CID suspends execution at the beginning of the statement containing the first COBOL verb. CID then gives you control.

The following considerations apply to the STEP command:

If a breakpoint or trap suspends program execution before CID has counted n lines, CID ignores the STEP command; execution is not suspended when n lines have been executed.

If a breakpoint or trap suspends program execution at the same time as CID has counted n lines, CID ignores the STEP command and displays the breakpoint or trap message.

The STEP command can also be used to execute your program until n procedure-name lines have been counted. The format of this command is as follows:

```
    STEP,n,PROCEDURE
```

When n procedure-name lines have been counted, CID issues the message:

```
    *S PROCEDURE AT PR.procedure-name
```

If the procedure-name is a paragraph within a section, CID displays only the name of the paragraph. CID suspends execution at the beginning of the procedure.

If the STEP command is entered with no parameters, the previous STEP command is repeated. If no previous STEP command has been entered, the default is STEP,1,LINE.

ALTERING EXECUTION FLOW

Sometimes it is useful to alter the flow of program execution. For example, you might want to reexecute a COBOL procedure without reexecuting the statements that precede that procedure, or you might want to pass over a paragraph or section that you know is faulty. The GO TO command alters execution flow.

The GO TO command resumes execution at the beginning of a paragraph or section. This command has the same form as the COBOL format 1 GO TO statement:

```
    GO TO procedure-name
or
    GO procedure-name
```

where procedure-name is the name of the paragraph or section where execution is to resume.

You should be careful when you enter this command, because altering execution flow changes your program logic: program values and file positions are not necessarily the same when you use this command to execute a paragraph as they are when execution reaches the paragraph normally.

NOTE

> You should only enter this command to resume execution (not to initiate execution). Program initialization that must take place will not occur if you begin a debug session with this command.

Examples of the GO TO command are:

```
    GO TO COUNT-ONE
```

This command resumes execution at the beginning of the COUNT-ONE paragraph or section.

```
    GO TO COUNT-ONE OF COUNT-ALL
```

This command resumes execution at the beginning of the COUNT-ONE paragraph in the COUNT-ALL section.

A sample debug session using the GO TO command is shown in figure 3-5. The program FIND-HIGH-BID with input file BIDS (both shown in section 3) are executed under CID control. The GO TO command is used to skip over the SORTING section causing the bids not to be sorted.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,pr.sorting
? go
 *B #1, AT PR.SORTING
? go to write-results ◄──────────── Skip the SORTING section and resume execution at the
                                    WRITE-RESULTS section.  Notice that the bids are not
                                    sorted.
        CUSTOMER      BID
        ID NUMBER

          CUST7      3344.22     <<HIGH BID

          CUST2      5544.62
          CUST8      3189.44
          CUST5      2266.44
          CUST3      9062.21
          CUST4      9111.32
          CUST9      0010.13

 *T #17, END IN L.160
? quit
 DEBUG TERMINATED
```

Figure 3-5. GO TO Command Example

SAMPLE DEBUG SESSION: ERRORS AND WARNINGS; SUSPENDING EXECUTION

The following paragraphs describe a sample debug session. Figure 3-6 shows the file BIDS used as input to the program FIND-HIGH-BID in figure 3-7. FIND-HIGH-BID is a more complicated version of the COBOL program presented in section 2. Each input record in BIDS contains a customer identification number and the amount of the bid submitted by that customer. FIND-HIGH-BID sorts the bids in descending order; the program prints the sorted list of customer identification numbers and bids, and indicates which bid is the highest. The program FIND-HIGH-BID contains no errors.

In figure 3-8, the program FIND-HIGH-BID is run under CID control. This debug session shows how traps, breakpoints, and the STEP command are used and how you respond to error and warning messages during a debug session.

```
Customer  Bid
   ID
CUST4334422
CUST2554462
CUST8318944
CUST5226644
CUST3906221
CUST4911132
CUST9001013
```

Figure 3-6. Input File BIDS

DISPLAYING PROGRAM VALUES

When execution of your program is suspended and CID has prompted you for input, you can enter commands to display program values as they exist at the time of suspension. This discussion shows the display commands that are most useful to the COBOL programmer.

CID provides two commands for displaying program values:

The LIST,VALUES command lists the values of all data-items in the program.

The DISPLAY command lists particular values that you specify in the command.

LIST,VALUES COMMAND

The LIST,VALUES command lists all the data names in the Data Division of the source program and the current values of each elementary data-item. Names are listed in the same order that they are specified in the Data Division. The LIST,VALUES command has the form:

```
LIST,VALUES
```

The LIST,VALUES command provides a formatted snapshot of the status of program variables; however, it can produce a large amount of output, particularly if the program contains large tables. To avoid a large amount of output, you can send the output to an auxiliary file as described later in this section or use the DISPLAY command to avoid large amounts of output. The short form of LIST,VALUES is LV.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  FIND-HIGH-BID.
*
*    THIS PROGRAM READS IN BIDS SUBMITTED FOR ONE ITEM
*    AT AN AUCTION AND SORTS THE BIDS IN DESCENDING
*    ORDER.  RESULTS ARE PRINTED OUT, AND THE HIGHEST
*    BID IS INDICATED.
*
*    EACH INPUT LINE TAKES THE FORM:
*      CUSTOMER-ID               PICTURE X(5).
*      BID                       PICTURE 9999V99.
*
 ENVIRONMENT DIVISION.

 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  CYBER-170.
 OBJECT-COMPUTER.  CYBER-170.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
     SELECT IN-FILE ASSIGN TO "BIDS".
     SELECT OUT-FILE ASSIGN TO "OUTPUT".
     SELECT SORT-FILE ASSIGN TO SFILE.

 DATA DIVISION.

 FILE SECTION.

 FD IN-FILE
     LABEL RECORD IS OMITTED
     DATA RECORD IS LINE-IN.

 01  LINE-IN.
     05  CUSTOMER-ID             PICTURE X(5).
     05  BID                     PICTURE 9999V99.
     05  FILLER                  PICTURE X(9).

 FD OUT-FILE
     LABEL RECORD IS OMITTED
     DATA RECORD IS LINE-OUT.

 01  LINE-OUT.
     05  FILLER                  PICTURE X(10).
     05  CUSTOMER-ID             PICTURE X(10).
     05  BID                     PICTURE 9999.99.
     05  HIGH-BID-OR-SPACES      PICTURE X(13).

 SD  SORT-FILE
     RECORD CONTAINS 11 CHARACTERS
     DATA RECORD IS SORT-RECORD.

 01  SORT-RECORD.
     05  BID                     PICTURE 9999V99.
     05  CUSTOMER-ID             PICTURE X(5).

```

Figure 3-7. Program FIND-HIGH-BID (Sheet 1 of 3)

```
WORKING-STORAGE SECTION.

01  SPACE-LINE                      PICTURE X(100) VALUE SPACES.

01  HEADING-1                       PICTURE X(25)
        VALUE "        CUSTOMER      BID".

01  HEADING-2                       PICTURE X(20)
        VALUE "        ID NUMBER".

01  HIGH-BID                        PICTURE X(13)
        VALUE "   <<HIGH BID".

01  BID-INFORMATION.
    05  NUMBER-OF-BIDS              PICTURE 9V.
    05  BID-TABLE                   OCCURS 10 TIMES
                                    INDEXED BY BID-INDEX.
        10  BID                     PICTURE 9999V99.
        10  CUSTOMER-ID             PICTURE X(5).

PROCEDURE DIVISION.

INITIALIZATION SECTION.

OPEN-FILES.
    OPEN INPUT IN-FILE
         OUTPUT OUT-FILE.
INITIALIZE-VALUES.
    MOVE ZERO TO NUMBER-OF-BIDS.

PROCESS-A-BID SECTION.

READ-A-BID.
    READ IN-FILE AT END GO TO SORTING.
    ADD 1 TO NUMBER-OF-BIDS.
    MOVE CORRESPONDING LINE-IN
            TO BID-TABLE (NUMBER-OF-BIDS).
READ-NEXT-BID.
    GO TO READ-A-BID.

SORTING SECTION.

SORT-THE-BIDS.
    SORT SORT-FILE
            ON DESCENDING KEY BID OF SORT-RECORD
          INPUT PROCEDURE IS SORT-IN-PROC
          OUTPUT PROCEDURE IS SORT-OUT-PROC.
DONE-SORTING.
    GO TO WRITE-RESULTS.

SORT-IN-PROC SECTION.

START-OF-SECTION.
    PERFORM SORTING-PARAGRAPH VARYING BID-INDEX FROM 1 BY 1
        UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
    GO TO END-OF-SECTION.
SORTING-PARAGRAPH.
    MOVE CORRESPONDING BID-TABLE (BID-INDEX)
          TO SORT-RECORD.
    RELEASE SORT-RECORD.
END-OF-SECTION.

```

Figure 3-7. Program FIND-HIGH-BID (Sheet 2 of 3)

```
119          SORT-OUT-PROC SECTION.
120
121          START-OF-SECTION.
122              PERFORM SORTING-PARAGRAPH VARYING BID-INDEX FROM 1 BY 1
123                  UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
124              GO TO END-OF-SECTION.
125          SORTING-PARAGRAPH.
126                RETURN SORT-FILE RECORD
127                    AT END GO TO END-OF-SECTION.
128                MOVE CORRESPONDING SORT-RECORD
129                        TO BID-TABLE (BID-INDEX).
130          END-OF-SECTION.
131
132          WRITE-RESULTS SECTION.
133          WRITE-HEADINGS.
134              WRITE LINE-OUT FROM SPACE-LINE.
135              WRITE LINE-OUT FROM SPACE-LINE.
136              WRITE LINE-OUT FROM HEADING-1.
137              WRITE LINE-OUT FROM HEADING-2.
138          WRITE-HIGH-BID.
139              WRITE LINE-OUT FROM SPACE-LINE.
140              MOVE HIGH-BID TO HIGH-BID-OR-SPACES OF LINE-OUT.
141              MOVE CORRESPONDING BID-TABLE (1) TO LINE-OUT.
142              WRITE LINE-OUT.
143              WRITE LINE-OUT FROM SPACE-LINE.
144          WRITE-REMAINING-BIDS.
145              PERFORM WRITE-A-BID VARYING BID-INDEX FROM 2 BY 1
146                  UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
147              GO TO END-WRITE.
148          WRITE-A-BID.
149              MOVE CORRESPONDING
150                    BID-TABLE (BID-INDEX) TO LINE-OUT.
151              WRITE LINE-OUT.
152          END-WRITE.
153              WRITE LINE-OUT FROM SPACE-LINE.
154
155          END-OF-RUN SECTION.
156
157          CLOSE-FILES.
158              CLOSE IN-FILE, OUT-FILE.
159          STOP-RUN.
160              STOP RUN.
161
```

Figure 3-7. Program FIND-HIGH-BID (Sheet 3 of 3)

```
 CYBER INTERACTIVE DEBUG
? set,trap,procedure,* <-------------------- Set PROCEDURE trap to suspend execution when-
? go                                          ever the beginning of a paragraph or section
 *T #1, PROCEDURE IN PR.INITIALIZATION        in the Procedure Division is reached.
? go
 *T #1, PROCEDURE IN PR.OPEN-FILES
? go
 *T #1, PROCEDURE IN PR.INITIALIZE-VALUES
? go
 *T #1, PROCEDURE IN PR.PROCESS-A-BID
? go
 *T #1, PROCEDURE IN PR.READ-A-BID
? list,trap,* <----------------------------- List the currently established traps.
 T #1 = PROCEDURE *
? clear,trap,* <---------------------------- Clear all traps.
? list,trap,*
 NO TRAPS                                     Set breakpoint at line 92.  The frequency
? set,breakpoint,l.92,1,50,2 <-------------- parameters indicate that the breakpoint
? set,breakpoint,pr.sort-the-bids             occurs every other time execution reaches
                                              line 92.
```

Figure 3-8. Sample Debug Session Using the Program FIND-HIGH-BID (Sheet 1 of 2)

```
? go
 *B #1, AT L.92
? display number-of-bids
  1
? go
 *B #1, AT L.92
? display number-of-bids
  3
? go                                                  Breakpoint number 1 occurs every other time
 *B #1, AT L.92                                       execution reaches line 92.
? display number-of-bids
  5
? go
 *B #1, AT L.92
? display number-of-bids
  7
? step,30,lines                                       Breakpoint number 2 suspends execution before
 *B #2, AT PR.SORT-THE-BIDS                           the STEP command has finished.  The STEP com-
? list,breakpoint,*                                   mand is no longer in effect.
 *B #1 = L.92,,50,2,   *B #2 = PR.SORT-THE-BIDS
? clear,breakpoint,#2                                 List all currently established breakpoints.
? list,breakpoint,*
 *B #1 = L.92,,50,2                                   Clear breakpoint number 2.
? step,1,procedure
 *S PROCEDURE AT PR.SORT-IN-PROC                      Resume execution until the beginning of the
? set,breakpoint,pr.write-results                     next paragraph or section is reached.
? set,trap,procedure,l.109...l.118
 *WARN - LINE 118 NOT EXECUTABLE - LINE 117 WILL BE USED    Set PROCEDURE trap with a range of lines as
 OK ? ok                                              the scope.
? go
 *T #1, PROCEDURE IN PR.START-OF-SECTION
? go
 *T #1, PROCEDURE IN PR.SORTING-PARAGRAPH
? go
 *T #1, PROCEDURE IN PR.SORTING-PARAGRAPH
? go
 *T #1, PROCEDURE IN PR.SORTING-PARAGRAPH
? go
 *T #1, PROCEDURE IN PR.SORTING-PARAGRAPH
? clear,trap,#1                                       Clear the PROCEDURE trap (trap number 1).
? go
 *B #2, AT PR.WRITE-RESULTS
? go


        CUSTOMER      BID
        ID NUMBER

          CUST4      9111.32    <<HIGH BID

          CUST3      9062.21
          CUST2      5544.62
          CUST7      3344.22
          CUST8      3189.44
          CUST5      2266.44
          CUST9      0010.13
                                                      The END trap occurs when the STOP RUN state-
 *T #17, END IN L.160                                 ment is executed.
? go
 *ERROR - PROGRAM HAS COMPLETED                       This error message tells you that execution
? quit                                                cannot be resumed in this manner after the
 DEBUG TERMINATED                                     END trap has occurred.
```

Figure 3-8. Sample Debug Session Using the Program FIND-HIGH-BID (Sheet 2 of 2)

An example of LIST,VALUES command output is shown in figure 3-9. The example is from a debug session executing the program FIND-HIGH-BID shown earlier in this section. Input is from the file BIDS, also shown earlier in this section.

The values listed in figure 3-9 were obtained when execution was suspended by a breakpoint set at the beginning of the WRITE-RESULTS section. The following points are illustrated in the figure:

LIST,VALUES output is formatted so that as much information as possible is displayed in as little space as possible. Although this format is difficult to read, the format increases the chances of fitting all of the output on a terminal screen.

Data names are displayed in the same order as they are specified in the Data Division. Every data name in the Data Division is displayed.

The level number of a data name is enclosed in a less than sign and a greater than sign; for example, the level number of a level 05 item appears as <05>. The level number appears to the left of the data name.

Index names are preceded by <IX>.

A group-item name is followed by a colon to indicate that the next item listed is part of that group-item.

An elementary data-item name is followed by an equal sign (=) and the value of the data-item.

Subscripts are enclosed in brackets ([]). The table name is displayed only for the first table element.

DISPLAY COMMAND

The DISPLAY command, introduced in section 2, is the most useful command for displaying COBOL program values. This command is a restricted form of the COBOL DISPLAY statement. The format is:

```
DISPLAY item-1, item-2,..., item-n
```

The items can be separated by commas or spaces and can be literals or identifiers.

The same types of identifiers can be specified in the DISPLAY command as can be specified in the COBOL DISPLAY statement:

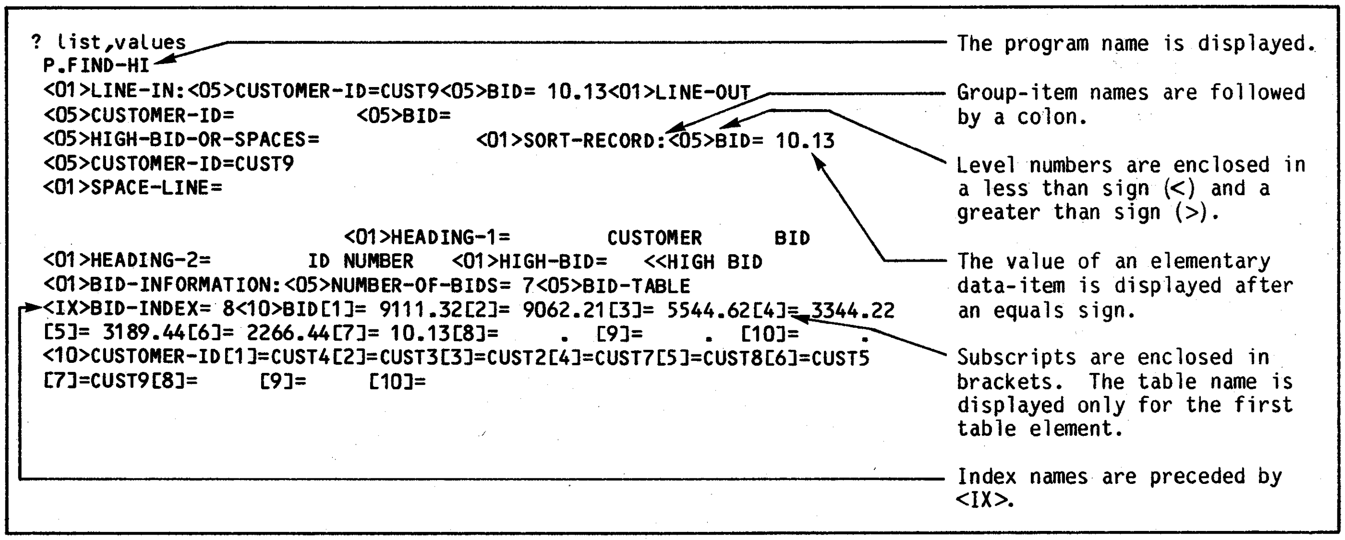
Identifiers specified in the DISPLAY command can be qualified or unqualified.

Subscripted table names can be specified to display specific elements.

Reference-modified table items can be specified.

Group-item names as well as the names of elementary data-items can be specified.

Examples of the DISPLAY command are shown in figure 3-10. The examples were obtained from the program FIND-HIGH-BID with input file BIDS, both shown earlier in this section. In the examples, execution is suspended at the beginning of the WRITE-RESULTS section. Program values are identical to those displayed earlier in the LIST,VALUES command example.



```
? list,values
 P.FIND-HI
 <01>LINE-IN:<05>CUSTOMER-ID=CUST9<05>BID= 10.13<01>LINE-OUT
 <05>CUSTOMER-ID=            <05>BID=
 <05>HIGH-BID-OR-SPACES=                  <01>SORT-RECORD:<05>BID= 10.13
 <05>CUSTOMER-ID=CUST9
 <01>SPACE-LINE=

                            <01>HEADING-1=        CUSTOMER      BID
 <01>HEADING-2=          ID NUMBER   <01>HIGH-BID=   <<HIGH BID
 <01>BID-INFORMATION:<05>NUMBER-OF-BIDS= 7<05>BID-TABLE
 <IX>BID-INDEX= 8<10>BID[1]= 9111.32[2]= 9062.21[3]= 5544.62[4]= 3344.22
 [5]= 3189.44[6]= 2266.44[7]= 10.13[8]=     .  [9]=     .  [10]=     .
 <10>CUSTOMER-ID[1]=CUST4[2]=CUST3[3]=CUST2[4]=CUST7[5]=CUST8[6]=CUST5
 [7]=CUST9[8]=      [9]=      [10]=
```

The program name is displayed.

Group-item names are followed by a colon.

Level numbers are enclosed in a less than sign (<) and a greater than sign (>).

The value of an elementary data-item is displayed after an equals sign.

Subscripts are enclosed in brackets. The table name is displayed only for the first table element.

Index names are preceded by <IX>.

Figure 3-9. LIST,VALUES Command Example

```
? display heading-2 <-------------------------------------------- Unqualified identifier
        ID NUMBER
? display number-of-bids of bid-information <-------------------- Qualified identifier
 7
? display customer-id of bid-table (2) <------------------------- Table element
 CUST3
? display customer-id of bid-table (2) (3:2) <------------------- Reference-modified table element
 ST
? display heading-2 (12:3) <------------------------------------- Reference-modified identifier
 NUM
? display bid-information <-------------------------------------- Group-item name
 7911132CUST4906221CUST3554462CUST2334422CUST7318944CUST8226644CUST500101
 3CUST9
? display bid-index <-------------------------------------------- Index name
  8
? display bid-index, number-of-bids, high-bid <------------------ List of identifiers
  8 7   <<HIGH BID
```

Figure 3-10. DISPLAY Command Examples

ALTERING PROGRAM VALUES

Once an error has occurred during a debug session, incorrect program values are usually present. At this point, you can terminate the debug session, correct and recompile your program, and start a new debug session to find the remaining errors. Alternatively, you can make a note of the error you found, alter the incorrect values to make them correct, and then resume execution of the same debug session to find the remaining errors. Altering program values does not actually correct the program; however, it does allow you to find more than one error during a debug session.

CID provides several commands to alter program values. This user's guide describes two CID commands that alter program values:

The MOVE command, which alters data items

The SET command, which alters index names and index data items

MOVE COMMAND

The MOVE command is a restricted form of the COBOL MOVE statement. The MOVE command has the form:

```
MOVE value TO identifier-1
```

A literal or identifier must be specified for value, and identifier-1 must be an acceptable receiving item. The following restrictions are placed on the MOVE command:

Only one receiving item is allowed.

MOVE CORRESPONDING is not allowed.

The sending and receiving items must be alphabetic, alphanumeric, numeric (other than COMP-1 or COMP-4) or group items. If either the sending or receiving item is COMP-2, both items must be COMP-2. Edited items are allowed, but they are moved without editing.

The sending item must not be a figurative constant.

The allowable forms of sending and receiving items are shown in table 3-3. If an unallowed combination of sending and receiving items is specified, CID issues an error message. The MOVE command functions exactly as in COBOL: identifier-1 receives the specified value. You can enter a MOVE command whenever CID has prompted you for input. For example, if program execution is suspended and you have detected a data-item that has an incorrect or illegal value, you can use the MOVE command to correct the value of the data-item. When you resume execution of the program, the new value is used in subsequent computations involving the altered data-item.

Changes made through the MOVE 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 data-items have the values defined in the original compiled version.

Following are some examples of the MOVE command:

```
MOVE 5 TO A
```

This command changes the value of A to five.

```
MOVE COUNT OF INPUT-LINE TO SIZE
```

This command changes the value of SIZE to the value of COUNT in the group-item INPUT-LINE.

```
MOVE NAME (15:10) TO FIRST-NAME
```

This command moves the ten characters beginning in the 15th character position of NAME to the data-item FIRST-NAME.

SET COMMAND

The SET command is a restricted form of the COBOL SET statement. This command is used to change the values of index-names during the course of a debug session.

The SET command has two formats. The format 1 SET command appears as follows:

```
SET name TO value
```

TABLE 3-3. ALLOWABLE MOVE COMMAND SENDING AND RECEIVING ITEMS

Sending Item	Receiving Item				
	Group	Alphabetic	Alphanumeric	Numeric	COMP-2
Group	Yes†	Yes	Yes	Yes	No
Alphabetic	Yes†	Yes	Yes	No	No
Alphanumeric	Yes†	Yes	Yes	Yes	No
Numeric	Yes†	No	Yes	Yes	No
COMP-2	No	No	No	No	Yes

†CID issues a warning message before this kind of move takes place.

In this format, name is an index-name or the identifier of an integer or index data-item; value is an integer literal, an index-name, or the identifier of an integer or index data-item.

Some combinations of sending-items and receiving-items are not allowed in the format 1 SET command. Table 3-4 shows the allowable combinations.

TABLE 3-4. ALLOWABLE FORMAT 1 SET COMMAND SENDING AND RECEIVING ITEMS

Sending Item	Receiving Item		
	Integer Data Item	Index Name	Index Data Item
Integer literal	No	Yes	No
Integer data item	No	Yes	No
Index name	Yes	Yes	Yes
Index data item	No	Yes	Yes

The format 2 SET command appears as follows:

```
    SET index UP BY amount
```

or

```
    SET index DOWN BY amount
```

In this format, the index is an index-name; amount is an integer literal or the identifier of an elementary numeric data-item. When you enter this command, the value of index is increased or decreased by the specified amount.

Following are examples of the two formats of the SET command:

SET INDEX-A TO 6

This command changes the value of INDEX-A to six.

SET INDEX-A UP BY 3

This command increases the value of INDEX-A by three.

SET BID-INDEX TO NUMBER-OF-BIDS

This command changes the value of BID-INDEX to the value of NUMBER-OF-BIDS.

SET FIRST-INDEX DOWN BY STEP-AMOUNT

This command decreases the value of FIRST-INDEX by the value of STEP-AMOUNT. In this case, STEP-AMOUNT must contain an integer value.

As with the MOVE command, changes that you make to index names do not exist beyond the end of the debug session. When a program is reexecuted, the index names have the values defined in the original program.

DISPLAYING CID AND PROGRAM STATUS INFORMATION

The following paragraphs describe 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; the values of these variables can be displayed at the terminal.

LIST commands that can display such things as load map information, and trap and breakpoint information.

DEBUG VARIABLES

CID provides 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 many of the debug variables are intended for use by assembly language programmers, some of them can provide information useful to COBOL programmers. Those variables that are most useful to COBOL programmers are listed in table 3-5. See the CID reference manual for a description of other debug variables.

TABLE 3-5. DEBUG VARIABLES

Variable	Description
#LINE	The number of the COBOL line executing at the time of suspension.
#PROC	The name of the paragraph or section executing at the time of suspension.
#BP	The number of defined breakpoints.
#TP	The number of defined traps.
#GP	The number of defined groups; groups are described in section 5.
#HOME	The name of the home program; the home program is the program unit currently being debugged. See section 4.

The #LINE debug variable contains the number of the COBOL source line that was executing at the time of suspension. The form of #LINE is P.progname_L.n, where the underscore (_) indicates a relative address in a program module or common block, progname is the name of the currently executing program, and n is the number of the currently executing line within that program. CID normally prints this information automatically when a trap or breakpoint occurs, but you might wish to display the value yourself at times.

The #PROC debug variable contains the name of the currently executing paragraph or section in the COBOL program. The form of #PROC is

```
    P.progname_PR.procedure-name
```

where progname is the name of the currently executing program; procedure-name is usually the name of the currently executing paragraph. When execution is suspended between a section name line and a paragraph name line, procedure-name is the name of the currently executing section. When execution is suspended within a paragraph that has the same name as a paragraph in a different section, #PROC does not indicate which of the two paragraphs was executing when suspension occurred.

The #HOME variable contains the name of the current home program. The home program is described in section 4. (When you are debugging a single main program, the home program is the program you are debugging.) The form of #HOME is P.name. This variable is useful for programs that contain multiple program units.

The #BP, #TP, and #GP variables contain the numbers of breakpoints, traps, and groups, respectively, that are currently defined for the debug session. These variables are especially useful for longer, more complex debug sessions.

To display the contents of a debug variable, you must use the D command; debug variables cannot be displayed with the DISPLAY command or LIST,VALUES command. The format of the D command is as follows:

```
    D,variable
```

In this format, variable is the name of the debug variable that you want to display. Only one variable can be specified in the D command.

NOTE

D is actually the short form of the command name DISPLAY. In the CID reference manual, this command is called the language-independent DISPLAY command; in this guide, it is called the D command so that it is not confused with the COBOL CID DISPLAY command that displays data items and literals. If you use the long form of the D command, you must enter a comma after the command name.

Examples of the D command are shown in figure 3-11. In this figure, the program FIND-HIGH-BID is executed with the input file BIDS. The program and input file are listed earlier in this section.

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 3-6.

TABLE 3-6. LIST COMMANDS

Command	Description
LIST,BREAKPOINT	Lists breakpoint information
LIST,GROUP	Lists group information
LIST,TRAP	Lists trap information
LIST,STATUS	Lists information about the current status of the debug session
LIST,VALUES	Lists all current program values

```
CYBER INTERACTIVE DEBUG
? set,breakpoint,pr.write-results
? go
 *B #1, AT PR.WRITE-RESULTS
? d,#line ◄──────────────────────────── #LINE is the line where execution is suspended.
 #LINE = P.FIND-HI_L.132
? d,#proc ◄──────────────────────────── #PROC is the paragraph or section where execution is suspended.
 #PROC = P.FIND-HI_PR.WRITE-RESULTS
? d,#bp ◄────────────────────────────── #BP is the number of currently defined breakpoints.
 #BP = 1
? list,breakpoint,*
 *B #1 = PR.WRITE-RESULTS
? d,#tp ◄────────────────────────────── #TP is the number of currently defined traps.
 #TP = 0
? list,trap,*
 NO TRAPS
? d,#gp ◄────────────────────────────── #GP is the number of currently defined groups.
 #GP = 0
? d,#home ◄──────────────────────────── #HOME is the name of the home program.
 #HOME = P.FIND-HI
? quit
```

Figure 3-11. Displaying Debug Variables

The LIST commands are particularly useful with longer debug sessions in which you are constantly changing the status of the session. For example, you can initially set some breakpoints or traps, clear some or all of them later in the session, and set new ones; or you can 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.

Some of the LIST commands can produce a large volume of output. You can prevent this output from appearing at the terminal by writing it, instead, to a separate file that can then be printed. The commands to accomplish this are described later in this section under Control of CID Output.

The LIST,BREAKPOINT and LIST,TRAP commands are described earlier in this section; the LIST,GROUP command is described in section 5. The LIST,STATUS command is described in the following paragraphs.

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

The short form of LIST,STATUS is LS. Information displayed by the LIST,STATUS command includes:

The home program name. The home program is described in section 4.

The number of breakpoints currently defined.

The number of traps currently defined.

The number of groups currently defined. Groups are described in section 5.

The status of veto mode (ON or OFF). Veto mode is described in the CID reference manual.

The status of interpret mode (ON or OFF). Interpret mode is described in the CID reference manual.

The current output options. Output options are controlled by the SET,OUTPUT command, which specifies the types of CID output sent to the terminal. The SET,OUTPUT command is described later in this section.

The current auxiliary file options. These options are specified by the SET,AUXILIARY command, which defines an auxiliary output file and specifies the type of output to be sent to that file. The SET,AUXILIARY command is described later in this section.

An example of the LIST,STATUS command is shown in figure 3-12.

CONTROL OF CID OUTPUT

The output produced by commands such as the LIST commands and the DISPLAY command can become voluminous. As an alternative to displaying all CID output at the terminal, you can define an auxiliary output file and specify certain types of CID output to be written to the file. The commands that control CID output are:

The SET,OUTPUT command, which specifies the types of output to be displayed at the terminal

The SET,AUXILIARY command, which defines an auxiliary output file and specifies the types of output to be sent to the file

For most debug sessions, these commands are not necessary.

```
? list,status
HOME = P.FIND-HI,   1 BREAKPOINTS,   NO TRAPS,   NO GROUPS,   VETO OFF
INTERPRET OFF,   OUT OPTIONS = I W E D,   AUXILIARY CLEAR
```

Figure 3-12. LIST,STATUS Command Example

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 one-letter code. The output codes, along with a description of each code, are listed in table 3-7.

TABLE 3-7. CID OUTPUT TYPES

Output Code	Description
E	Error messages.
W	Warning messages.
D	Output produced by execution of CID commands. Includes output produced by LIST, DISPLAY, D, and TRACBACK commands.
I	Informative messages. Includes breakpoint and trap messages.
R	The text of each command when it is executed from a group or file command sequence.
B	The text of each command when it is executed from a trap or breakpoint body.
T	The text of each command entered from the terminal.

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,type-list
```

where type-list is a list of output type codes as shown in table 3-7. The type codes can be separated by commas, or they can be entered without separators. The short form of SET,OUTPUT is SOUT.

If you include an output code in the option list of the SET,OUTPUT command, the associated output type is 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 it suppresses types D, R, and B.

When the list is omitted, the default options are E, W, D, and I for interactive jobs, and E, W, D, I, R, B, and T for batch jobs. If a SET,OUTPUT command is not entered, these output types are displayed at the terminal. It is unnecessary to specify type T in a SET,OUTPUT command, because terminal input is displayed at the terminal when you enter it.

The only output types not automatically displayed are commands executed in command sequences (types R and B). Command sequences are described in section 5. To display both this output and the default output types, enter the command:

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

If you specify the R option on the SET,OUTPUT command, then whenever a READ command is executed, each command in the specified group or file command sequence is displayed at the terminal as it is executed. If you specify the B option, whenever a breakpoint or trap body is executed, each command in the body is displayed as it is executed.

The only output types that cannot be suppressed are the informative messages issued when breakpoints or traps 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 action described in the warning message, responding as if you had entered a YES or OK response (described earlier in this section under Error and Warning Processing).

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

```
CLEAR,OUTPUT
```

The short form of CLEAR,OUTPUT is COUT.

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 the execution of CID commands). This includes output produced by the LIST,VALUES and DISPLAY commands, both 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,type-list
```

where lfn is the name of the auxiliary file and type-list is a list of output type codes as shown in table 3-7. The type codes can be separated by commas, or they can be entered without separators. The short form of SET,AUXILIARY is SAUX.

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 suppress these output types.

The option specifications for an auxiliary file can be changed simply by entering another SET,AUXILIARY command that specifies 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, enter the command:

```
CLEAR,AUXILIARY
```

The short form of CLEAR,AUXILIARY is CAUX. An auxiliary file can be closed at any time during a debug session.

The auxiliary file is a local file. After you terminate the debug session, you can display the auxiliary file at the terminal, send it to a printer, or store 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 subsequent sessions, 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. For example, the command

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

writes the output types E, W, D, I, and T to file OUTF. If you enter this command at the beginning of a debug session, a copy of the session is created exactly as displayed at the terminal. Note that when you are using an auxiliary file, you must specify the T option to include in the file the commands you entered.

The example in figure 3-13 illustrates a SET,OUTPUT command used in conjunction with a SET,AUXILIARY command to suppress output to the terminal and write it instead to an auxiliary file. In this example, the program FIND-HIGH-BID was executed under CID control with the input file BIDS. In the figure, execution is suspended at the beginning of the WRITE-RESULTS section. The program FIND-HIGH-BID and the input file BIDS are shown earlier in this section.

```
? set,output,e,w,i
? set,auxiliary,values,d
? list,values
? clear,auxiliary
? set,output,e,w,i,d
```

Figure 3-13. Output Options Example

The example in figure 3-13 suppresses all output produced by CID commands (output type D), creates an auxiliary file called VALUES to which this output is to be written, writes all program values to the file VALUES, closes VALUES, and resets output options to original conditions. The resulting file VALUES is shown in figure 3-14.

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

```
?SET,AUXILIARY,AUXF,D,E
?CLEAR,OUTPUT
```

This example defines an auxiliary file named AUXF to receive error messages and output from CID commands and turns off output to the terminal (except for trap and breakpoint messages). To close AUXF and restore terminal output to the default conditions, you can enter:

```
?SET,OUTPUT,E,W,D,I
?CLEAR,AUXILIARY
```

```
1                                                  CYBER INTERACTIVE DEBUG 1.2-552.
0
 P.FIND-HI
 <01>LINE-IN:<05>CUSTOMER-ID=CUST9<05>BID= 10.13<01>LINE-OUT
 <05>CUSTOMER-ID=          <05>BID=
 <05>HIGH-BID-OR-SPACES=             <01>SORT-RECORD:<05>BID= 10.13
 <05>CUSTOMER-ID=CUST9
 <01>SPACE-LINE=

                            <01>HEADING-1=       CUSTOMER      BID
 <01>HEADING-2=        ID NUMBER   <01>HIGH-BID=    <<HIGH BID
 <01>BID-INFORMATION:<05>NUMBER-OF-BIDS= 7<05>BID-TABLE
 <IX>BID-INDEX= 8<10>BID[1]= 9111.32[2]= 9062.21[3]= 5544.62[4]= 3344.22
 [5]= 3189.44[6]= 2266.44[7]= 10.13[8]=      .  [9]=     .  [10]=     .
 <10>CUSTOMER-ID[1]=CUST4[2]=CUST3[3]=CUST2[4]=CUST7[5]=CUST8[6]=CUST5
 [7]=CUST9[8]=      [9]=      [10]=
```

Figure 3-14. Auxiliary Output File VALUES

INTERACTIVE INPUT

Programs receiving input from the terminal can be executed under CID control. A program that is to receive input from the terminal should be written 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 you are running programs under NOS since the system automatically inserts a question mark (?) prompt (identical to the CID prompt) at the beginning of a line to indicate a program request for terminal input.

In figure 3-15, the program FIND-HIGH-BID has been modified to accept input from the terminal. The following modifications were made to form the new program FIND-HIGH-BID-2:

Line 22 was changed to reference the file INPUT rather than file BIDS.

Lines 60 through 66 were inserted so that the program could request input from you and the program could prompt you for input using a greater than sign (>).

Lines 98 and 99 were inserted to make the program issue the input request line before the list of bids is entered from the terminal.

Line 101 was inserted to make the program display the greater than sign before each input line is entered from the terminal.

```
 1        IDENTIFICATION DIVISION.
 2        PROGRAM-ID.  FIND-HIGH-BID-2.
 3       *
 4       *    THIS PROGRAM READS IN BIDS SUBMITTED FOR ONE ITEM
 5       *    AT AN AUCTION, AND SORTS THE BIDS IN DESCENDING
 6       *    ORDER.  RESULTS ARE PRINTED OUT, AND THE HIGHEST
 7       *    BID IS INDICATED.
 8       *
 9       *    EACH INPUT LINE TAKES THE FORM:
10       *      CUSTOMER-ID              PICTURE X(5).
11       *      BID                      PICTURE 9999V99.
12       *
13        ENVIRONMENT DIVISION.
14
15        CONFIGURATION SECTION.
16        SOURCE-COMPUTER.  CYBER-170.
17        OBJECT-COMPUTER.  CYBER-170.
18
```

Figure 3-15. Program FIND-HIGH-BID-2 (Sheet 1 of 4)

```
INPUT-OUTPUT SECTION.

FILE-CONTROL.
    SELECT IN-FILE ASSIGN TO "INPUT".
    SELECT OUT-FILE ASSIGN TO "OUTPUT".
    SELECT SORT-FILE ASSIGN TO SFILE.

DATA DIVISION.

FILE SECTION.

FD IN-FILE
    LABEL RECORD IS OMITTED
    DATA RECORD IS LINE-IN.

01  LINE-IN.
    05  CUSTOMER-ID             PICTURE X(5).
    05  BID                     PICTURE 9999V99.
    05  FILLER                  PICTURE X(9).

FD OUT-FILE
    LABEL RECORD IS OMITTED
    DATA RECORD IS LINE-OUT.

01  LINE-OUT.
    05  FILLER                  PICTURE X(10).
    05  CUSTOMER-ID             PICTURE X(10).
    05  BID                     PICTURE 9999.99.
    05  HIGH-BID-OR-SPACES      PICTURE X(13).
    05  FILLER                  PICTURE X(30).

SD  SORT-FILE
    RECORD CONTAINS 11 CHARACTERS
    DATA RECORD IS SORT-RECORD.

01  SORT-RECORD.
    05  BID                     PICTURE 9999V99.
    05  CUSTOMER-ID             PICTURE X(5).

WORKING-STORAGE SECTION.

01  INPUT-REQUEST-LINE.
    05  PART-1                  PICTURE X(37)
         VALUE " ENTER LIST OF CUSTOMER-IDS AND BIDS ".
    05  PART-2                  PICTURE X(30)
         VALUE "IN THE FORM XXXXX9999V99".

01  PROMPT-LINE                 PICTURE X(3) VALUE " > ".

01  SPACE-LINE                  PICTURE X(100) VALUE SPACES.

01  HEADING-1                   PICTURE X(25)
        VALUE "        CUSTOMER      BID".

01  HEADING-2                   PICTURE X(20)
        VALUE "        ID NUMBER".

01  HIGH-BID                    PICTURE X(13)
        VALUE "   <<HIGH BID".

01  BID-INFORMATION.
    05  NUMBER-OF-BIDS          PICTURE 9V.
    05  BID-TABLE               OCCURS 10 TIMES
                                INDEXED BY BID-INDEX.
        10  BID                 PICTURE 9999V99.
        10  CUSTOMER-ID         PICTURE X(5).

```

Figure 3-15. Program FIND-HIGH-BID-2 (Sheet 2 of 4)

```
PROCEDURE DIVISION.

INITIALIZATION SECTION.

OPEN-FILES.
    OPEN INPUT IN-FILE
         OUTPUT OUT-FILE.
INITIALIZE-VALUES.
    MOVE ZERO TO NUMBER-OF-BIDS.

PROCESS-A-BID SECTION.

REQUEST-INPUT.
    WRITE LINE-OUT FROM INPUT-REQUEST-LINE.
READ-A-BID.
    WRITE LINE-OUT FROM PROMPT-LINE.
    READ IN-FILE AT END GO TO SORTING.
    ADD 1 TO NUMBER-OF-BIDS.
    MOVE CORRESPONDING LINE-IN
            TO BID-TABLE (NUMBER-OF-BIDS).
READ-NEXT-BID.
    GO TO READ-A-BID.

SORTING SECTION.

SORT-THE-BIDS.
    SORT SORT-FILE
            ON DESCENDING KEY BID OF SORT-RECORD
          INPUT PROCEDURE IS SORT-IN-PROC
          OUTPUT PROCEDURE IS SORT-OUT-PROC.
DONE-SORTING.
    GO TO WRITE-RESULTS.

SORT-IN-PROC SECTION.

START-OF-SECTION.
    PERFORM SORTING-PARAGRAPH VARYING BID-INDEX FROM 1 BY 1
        UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
    GO TO END-OF-SECTION.
SORTING-PARAGRAPH.
    MOVE CORRESPONDING BID-TABLE (BID-INDEX)
          TO SORT-RECORD.
    RELEASE SORT-RECORD.
END-OF-SECTION.

SORT-OUT-PROC SECTION.

START-OF-SECTION.
    PERFORM SORTING-PARAGRAPH VARYING BID-INDEX FROM 1 BY 1
        UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
    GO TO END-OF-SECTION.
SORTING-PARAGRAPH.
      RETURN SORT-FILE RECORD
          AT END GO TO END-OF-SECTION.
      MOVE CORRESPONDING SORT-RECORD
              TO BID-TABLE (BID-INDEX).
END-OF-SECTION.
```

Figure 3-15. Program FIND-HIGH-BID-2 (Sheet 3 of 4)

```
144          WRITE-RESULTS SECTION.
145          WRITE-HEADINGS.
146              WRITE LINE-OUT FROM SPACE-LINE.
147              WRITE LINE-OUT FROM SPACE-LINE.
148              WRITE LINE-OUT FROM HEADING-1.
149              WRITE LINE-OUT FROM HEADING-2.
150          WRITE-HIGH-BID.
151              WRITE LINE-OUT FROM SPACE-LINE.
152              MOVE HIGH-BID TO HIGH-BID-OR-SPACES OF LINE-OUT.
153              MOVE CORRESPONDING BID-TABLE (1) TO LINE-OUT.
154              WRITE LINE-OUT.
155              WRITE LINE-OUT FROM SPACE-LINE.
156          WRITE-REMAINING-BIDS.
157              PERFORM WRITE-A-BID VARYING BID-INDEX FROM 2 BY 1
158                  UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
159              GO TO END-WRITE.
160          WRITE-A-BID.
161              MOVE CORRESPONDING
162                     BID-TABLE (BID-INDEX) TO LINE-OUT.
163              WRITE LINE-OUT.
164          END-WRITE.
165              WRITE LINE-OUT FROM SPACE-LINE.
166
167          END-OF-RUN SECTION.
168
169          CLOSE-FILES.
170              CLOSE IN-FILE, OUT-FILE.
171          STOP-RUN.
172              STOP RUN.
173
```

Figure 3-15. Program FIND-HIGH-BID-2 (Sheet 4 of 4)

A debug session for FIND-HIGH-BID-2 using NOS is shown in figure 3-16. Notice in the figure that question mark prompts for input to the program can be confused with prompts for CID commands. The greater than signs printed just before the input lines tell you that the input is being requested by the program rather than by CID. When you are debugging a program that requests terminal input you should carefully read messages that are displayed at the terminal to see if the input request is from your program or from CID.

A debug session for FIND-HIGH-BID-2 using NOS/BE is shown in figure 3-17. NOS/BE gives no prompt for input; the greater than sign is therefore important in the program using NOS/BE, because it prompts you directly for program input. Without this prompt, it can be difficult to determine whether you should enter program input or wait for either CID output or program output when the terminal is inactive.

SAMPLE DEBUG SESSION: DISPLAYING AND ALTERING VALUES; OUTPUT CONTROL

The program FIND-HIGH-BID and its associated input file BIDS (both shown earlier in this section) are used for the sample debug session described in the following paragraphs. The debug session is shown in figure 3-18; the auxiliary file created during the debug session is shown in figure 3-19.

In this debug session, a breakpoint is set at the beginning of the SORTING section and execution is initiated. When the breakpoint occurs, all seven of the bids on file BIDS have been read by the program. Using MOVE commands, an eighth bid is given to the program while execution is suspended.

At this point, an auxiliary file AUXFILE is created with the E, W, I, D, and T output options, and the standard output file is changed so that the D option is not in effect. Error messages, warning messages, and informative messages are written to both files. Commands entered from the terminal are written to the auxiliary file. CID output is written to the auxiliary file only. The debug variables are then displayed on the auxiliary file (see figure 3-19) to document on the file where execution was suspended when the file was created. When the LIST,VALUES command is entered, values are listed on the auxiliary file and not on the terminal. The auxiliary file is then closed by the CLEAR,AUXILIARY command, and the D output option is reinstated for the standard output file by the SET,OUTPUT command.

Next, a procedure trap is set with lines 99 through 106 as the scope. Because the scope is restricted to these lines only, this trap can suspend execution only when the SORT-THE-BIDS and DONE-SORTING paragraphs are reached. A breakpoint is then set at the beginning of the WRITE-RESULTS section, and execution is resumed.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,pr.read-next-bid
? go
 ENTER LIST OF CUSTOMER-IDS AND BIDS IN THE FORM XXXXX9999V99
 >
? cus0452144 <──────────────────────────────────────────────── Program input
 *B #1, AT PR.READ-NEXT-BID
? display bid of bid-table (number-of-bids)
  521.44
? go
 >
? cus71061318 <─────────────────────────────────────────────── Program input
 *B #1, AT PR.READ-NEXT-BID
? display bid of bid-table (number-of-bids)
  613.18
? go
 >
? cus08042500 <─────────────────────────────────────────────── Program input
 *B #1, AT PR.READ-NEXT-BID
? display bid of bid-table (number-of-bids)
  425.00
? go
 >
? cus97071133 <─────────────────────────────────────────────── Program input
 *B #1, AT PR.READ-NEXT-BID
? display bid of bid-table (number-of-bids)
  711.33
? go
 >
? cus36055400 <─────────────────────────────────────────────── Program input
 *B #1, AT PR.READ-NEXT-BID
? display bid of bid-table (number-of-bids)
  554.00
? go
 >
?


        CUSTOMER      BID
        ID NUMBER

          CUS97     0711.33   <<HIGH BID

          CUS71     0613.18
          CUS36     0554.00
          CUS04     0521.44
          CUS08     0425.00

 *T #17, END IN L.172
?
 DEBUG TERMINATED
```

Figure 3-16. Interactive Program Input (NOS)

```
CYBER INTERACTIVE DEBUG
?set,breakpoint,pr.read-next-bid
?go
ENTER LIST OF CUSTOMER-IDS AND BIDS IN THE FORM XXXXX9999V99
>cus04052144 <------------------------------------- Program input
*B #1, AT PR.READ-NEXT-BID
?display bid of bid-table (number-of-bids)
 521.44
?go
>cus71061318 <------------------------------------- Program input
*B #1, AT PR.READ-NEXT-BID
?display bid of bid-table (number-of-bids)
 613.18
?go
>cus08042500 <------------------------------------- Program input
*B #1, AT PR.READ-NEXT-BID
?display bid of bid-table (number-of-bids)
 425.00
?go
>cus97071133 <------------------------------------- Program input
*B #1, AT PR.READ-NEXT-BID
?display bid of bid-table (number-of-bids)
 711.33
?go
>cus36055400 <------------------------------------- Program input
*B #1, AT PR.READ-NEXT-BID
?display bid of bid-table (number-of-bids)
 554.00
?go
>%eof


       CUSTOMER      BID
       ID NUMBER

         CUS97      0711.33    <<HIGH BID

         CUS71      0613.18
         CUS36      0554.00
         CUS04      0521.44
         CUS08      0425.00

*T #17, END IN L.172
?quit
  DEBUG TERMINATED
```

Figure 3-17. Interactive Program Input (NOS/BE)

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,pr.sorting
? go
 *B #1, AT PR.SORTING
? display number-of-bids
  7
? move 8 to number-of-bids                        }
? move "cust6" to customer-id of bid-table (8)    }
? move 596.25 to bid of bid-table (8)             }
? set,output,e,w,i
? set,auxiliary,auxfile,e,w,i,d,t
? d,#proc
? d,#line
? list,values
```

Display the value of NUMBER-OF-BIDS.

Through MOVE commands, place an additional bid in BID-INFORMATION.

Because no D option is specified, CID output will not appear at the terminal.

Establish the auxiliary file AUXFILE. The output options in the SET,OUTPUT and SET,AUXILIARY commands cause CID output to be sent to the auxiliary file, but not to the terminal.

Figure 3-18. Sample Debug Session Using the Program FIND-HIGH-BID (Sheet 1 of 3)

```
? set,output,e,w,i,d
? clear,auxiliary
? set,trap,procedure,l.99...l.106
 *WARN - LINE 106 NOT EXECUTABLE - LINE 105 WILL BE USED
 OK ? ok
? set,breakpoint,pr.write-results
? go
 *T #1, PROCEDURE IN PR.SORT-THE-BIDS
? go
 *T #1, PROCEDURE IN PR.DONE-SORTING
? go
 *B #2, AT PR.WRITE-RESULTS
? set,auxiliary,auxfile,e,w,i,d,t
? d,#proc
 #PROC = P.FIND-HI_PR.WRITE-RESULTS
? d,#line
 #LINE = P.FIND-HI_L.132
? list,status
 HOME = P.FIND-HI,   2 BREAKPOINTS,   1 TRAPS,   NO GROUPS,   VETO OFF
 INTERPRET OFF,   OUT OPTIONS = I W E D
 AUX FILE = AUXFILE, OPTIONS = I W E D T
? set,output,e,w,i
? list,values
? list,trap,*
? list,breakpoint,*
? set,breakpoint,pr.write-a-bid
? go


        CUSTOMER      BID
        ID NUMBER

          CUST4     9111.32    <<HIGH BID

 *B #3, AT PR.WRITE-A-BID
? go
          CUST3     9062.21
 *B #3, AT PR.WRITE-A-BID
? go
          CUST2     5544.62
 *B #3, AT PR.WRITE-A-BID
? go
          CUST7     3344.22
 *B #3, AT PR.WRITE-A-BID
? display bid-index
? set bid-index down by 2
? go
          CUST2     5544.62
 *B #3, AT PR.WRITE-A-BID
? go
          CUST7     3344.22
 *B #3, AT PR.WRITE-A-BID
? go
          CUST8     3189.44
 *B #3, AT PR.WRITE-A-BID
? display bid-index
? set bid-index to 3
? go
          CUST2     5544.62
 *B #3, AT PR.WRITE-A-BID
? clear,auxiliary
? step,1,line
 *S LINE AT L.149
? step,1,line
 *S LINE AT L.151
? step,1,line
          CUST7     3344.22
 *B #3, AT PR.WRITE-A-BID
```

Specify the D option to resume sending CID output to the terminal. (? set,output,e,w,i,d)

Close the auxiliary file. (? clear,auxiliary)

Establish AUXFILE again as the auxiliary output file. Output is appended to the end of the file. (? set,auxiliary,auxfile,e,w,i,d,t)

The debug variables are displayed both at the terminal and on the auxiliary file, because the D output option is in effect for both the standard and auxiliary output files. (? d,#proc … #LINE = P.FIND-HI_L.132)

The D option is suppressed from the standard output file. CID output will not appear at the terminal. (? set,output,e,w,i)

The format 2 SET command alters BID-INDEX. (? set bid-index down by 2)

The format 1 SET command alters BID-INDEX. (? set bid-index to 3)

Close the auxiliary output file. (? clear,auxiliary)

Figure 3-18. Sample Debug Session Using the Program FIND-HIGH-BID (Sheet 2 of 3)

```
? d,#tp
? set,output,e,w,i,d,t
? d,#tp
 #TP = 1
? d,#bp
 #BP = 3
? list,breakpoint,*
 *B #1 = PR.SORTING,   *B #2 = PR.WRITE-RESULTS,   *B #3 = PR.WRITE-A-BID
? clear,breakpoint,#3
? d,#bp
 #BP = 2
? go
          CUST8     3189.44
          CUST5     2266.44
          CUST6     0596.25
          CUST9     0010.13

 *T #17, END IN L.160
? quit
 DEBUG TERMINATED
```

#TP is not displayed, because the D option is not in effect for the standard output file.

The D option is specified in the SET,OUTPUT command so that #TP can be displayed.

#BP is decreased by one when breakpoint number 3 is cleared.

Figure 3-18. Sample Debug Session Using the Program FIND-HIGH-BID (Sheet 3 of 3)

```
1                                                  CYBER INTERACTIVE DEBUG 1.2-552.
0
 D,#PROC
 #PROC = P.FIND-HI_PR.SORTING
 D,#LINE
 #LINE = P.FIND-HI_L.97
 LIST,VALUES
 P.FIND-HI
 <01>LINE-IN:<05>CUSTOMER-ID=CUST9<05>BID= 10.13<01>LINE-OUT
 <05>CUSTOMER-ID=          <05>BID=
 <05>HIGH-BID-OR-SPACES=             <01>SORT-RECORD:<05>BID=     .
 <05>CUSTOMER-ID=
 <01>SPACE-LINE=

                              <01>HEADING-1=        CUSTOMER     BID
 <01>HEADING-2=        ID NUMBER   <01>HIGH-BID=   <<HIGH BID
 <01>BID-INFORMATION:<05>NUMBER-OF-BIDS= 8<05>BID-TABLE
 <IX>BID-INDEX=-306783378<10>BID[1]= 3344.22[2]= 5544.62[3]= 3189.44[4]=
  2266.44[5]= 9062.21[6]= 9111.32[7]= 10.13[8]= 596.25[9]=     .  [10]=
     .  <10>CUSTOMER-ID[1]=CUST7[2]=CUST2[3]=CUST8[4]=CUST5[5]=CUST3[6]=
 CUST4[7]=CUST9[8]=CUST6[9]=     [10]=
 SET,OUTPUT,E,W,I,D
 CLEAR,AUXILIARY
```

The SET,AUXILIARY command is specified when execution is suspended at line 97, the beginning of the SORTING section. CID output and commands entered at the terminal are listed, because the D and T options are specified in the SET,AUXILIARY command.

The auxiliary output file is closed. Output is not sent to this file until the next SET,AUXILIARY command is entered.

Figure 3-19. Auxiliary Output File AUXFILE (Sheet 1 of 2)

```
1
0                                                CYBER INTERACTIVE DEBUG 1.2-552.
 D,#PROC                                                                    A new heading is written
 #PROC = P.FIND-HI_PR.WRITE-RESULTS                                         to the auxiliary file when
 D,#LINE                                                                    the SET,AUXILIARY command
 #LINE = P.FIND-HI_L.132                                                    is entered.
 LIST,STATUS
 HOME = P.FIND-HI,   2 BREAKPOINTS,   1 TRAPS,   NO GROUPS,   VETO OFF
 INTERPRET OFF,   OUT OPTIONS = I W E D
 AUX FILE = AUXFILE, OPTIONS = I W E D T
 SET,OUTPUT,E,W,I
 LIST,VALUES
 P.FIND-HI
 <01>LINE-IN:<05>CUSTOMER-ID=CUST9<05>BID= 10.13<01>LINE-OUT
 <05>CUSTOMER-ID=           <05>BID=
 <05>HIGH-BID-OR-SPACES=               <01>SORT-RECORD:<05>BID= 10.13
 <05>CUSTOMER-ID=CUST9
 <01>SPACE-LINE=

                             <01>HEADING-1=        CUSTOMER      BID
 <01>HEADING-2=        ID NUMBER   <01>HIGH-BID=    <<HIGH BID
 <01>BID-INFORMATION:<05>NUMBER-OF-BIDS= 8<05>BID-TABLE
 <IX>BID-INDEX= 9<10>BID[1]= 9111.32[2]= 9062.21[3]= 5544.62[4]= 3344.22
 [5]= 3189.44[6]= 2266.44[7]= 596.25[8]= 10.13[9]=     .  [10]=     .
 <10>CUSTOMER-ID[1]=CUST4[2]=CUST3[3]=CUST2[4]=CUST7[5]=CUST8[6]=CUST5
 [7]=CUST6[8]=CUST9[9]=      [10]=
 LIST,TRAP,*
 T #1 = PROCEDURE L.99...L.106
 LIST,BREAKPOINT,*
 *B #1 = PR.SORTING,   *B #2 = PR.WRITE-RESULTS
 SET,BREAKPOINT,PR.WRITE-A-BID
 GO
 *B #3, AT PR.WRITE-A-BID                                                   Program output does not
 GO                                                                         appear on the auxiliary
 *B #3, AT PR.WRITE-A-BID                                                   file.
 GO
 *B #3, AT PR.WRITE-A-BID
 GO
 *B #3, AT PR.WRITE-A-BID
 DISPLAY BID-INDEX
  5
 SET BID-INDEX DOWN BY 2
 GO
 *B #3, AT PR.WRITE-A-BID
 GO
 *B #3, AT PR.WRITE-A-BID
 GO
 *B #3, AT PR.WRITE-A-BID
 DISPLAY BID-INDEX
  6
 SET BID-INDEX TO 3
 GO
 *B #3, AT PR.WRITE-A-BID
 CLEAR,AUXILIARY                                                            The auxiliary file is
                                                                            closed.
```

Figure 3-19. Auxiliary Output File AUXFILE (Sheet 2 of 2)

When execution is suspended at the beginning of the WRITE-RESULTS section, the auxiliary file AUXFILE is again established with the E, W, I, D, and T options. #PROC and #LINE are again displayed to document on the auxiliary file where execution is suspended. The debug variables are also displayed at the terminal, because the D option has not been suppressed. The status of the debug session is then listed on both the auxiliary file and at the terminal.

After the status is listed, the SET,OUTPUT command is entered to suppress CID output at the terminal. Program values, traps, and breakpoints are listed on the auxiliary file, and a breakpoint is set at the WRITE-A-BID paragraph. Execution is resumed.

Execution is suspended and resumed several times at the WRITE-A-BID paragraph. When BID-INDEX is equal to five, its value is displayed on the auxiliary file. The format 2 SET command is entered to decrease BID-INDEX by two; this action causes two output lines to be repeated. When BID-INDEX is six, the format 1 SET command changes the value to 3, and several output lines are again repeated. Notice that program output does not appear on the auxiliary file. The next time execution is suspended, the auxiliary file is closed.

The STEP command is then entered several times. The first time STEP,1,LINE is entered, execution is suspended at line 149. The second time STEP,1,LINE is entered, execution is suspended at line 151, two lines after line 149. Line 150 is skipped, because line 150 is continued from line 149. The third time STEP,1,LINE is entered, execution is suspended by the breakpoint at the WRITE-A-BID paragraph instead of by the STEP command. When a breakpoint and STEP command cause execution to be suspended at the same line, the breakpoint suspends execution and the STEP command is ignored.

When execution is suspended, an attempt is made to display #TP, using the D command. #TP is not displayed, however, because the D option has been suppressed from the standard output file. The SET,OUTPUT command is entered to reinstate the D option, and then #TP is displayed. One trap is currently established.

Next, #BP is displayed, and three breakpoints are established. The three breakpoints are listed and breakpoint number three is cleared. #BP decreases to two as a result of this clear command.

Execution is resumed. The program executes to completion.

MULTIPLE PROGRAM UNITS 4

This section describes CID features for debugging COBOL programs that call one or more subprograms. In this user's guide, the main program and each subprogram is called a program unit.

Special features are necessary when you are debugging a main program with subprograms, because identifiers, line numbers, and procedure-names in a program unit are local to that program unit. Different program units might have some of the same line numbers, procedure-names, and identifiers. CID allows you to specify particular program units in many commands, and CID issues program unit information when program execution is suspended in different program units.

HOME PROGRAM

The home program is a program unit designated by CID or by you. Whenever execution is suspended by a breakpoint, a trap, or the STEP command, CID automatically designates the suspended program unit as the home program. You can designate the home program explicitly by entering the SET,HOME command, but the next time execution is suspended, the home program reverts to the suspended program unit.

When you refer to line numbers and procedure-names in the home program, no special notation is necessary. However, when you refer to line numbers and procedure-names that are outside the home program, you must use program name qualification, described later in this section.

CID commands with a COBOL syntax can only specify identifiers and procedure names inside the home program. The CID commands of this type that are described in this user's guide are as follows:

DISPLAY command

GO TO command

MOVE command

SET command

If you want to enter one of these commands and specify an identifier outside the home program, you must first enter the SET,HOME command to designate the program unit containing the identifier as the home program.

The following paragraphs describe the SET,HOME command and the #HOME debug variable. An example debug session is shown in figure 4-1 to illustrate these concepts. The session is run using the main program PROCESS-BIDS (figure 4-2), which calls the subprogram SORT-THE-BIDS (figure 4-3). The input file BIDS is shown in figure 4-4.

SET,HOME COMMAND

To explicitly designate the home program, you enter the SET,HOME command:

```
SET,HOME,program-unit
```

where program-unit specifies which program unit is the home program. Only the first seven characters in the program unit name need to be specified. The short form of SET,HOME is SH.

Examples of the SET,HOME command are:

```
SET,HOME,PREPARE-PAYROLL
```

This command designates PREPARE-PAYROLL as the home program.

```
SET,HOME,PREPARE
```

This command has the same effect as the previous example, because only the first seven characters in the program unit name need to be specified.

```
SH,PREPARE
```

This command has the same effect as the previous example.

#HOME DEBUG VARIABLE

The debug variable #HOME contains the first seven characters of the name of the current home program. You can display the value of #HOME by entering the D command as follows:

```
D,#HOME
```

The D command is described in section 3.

```
CYBER INTERACTIVE DEBUG
? set,breakpoint,pr.sorting
? go
 *B #1, AT PR.SORTING
? d,#home                               Execution is suspended in PROCESS-BIDS.  Therefore, PROCESS-BIDS
 #HOME = P.PROCESS                      is designated the home program.  #HOME contains the first seven
? display number-of-bids                characters of the home program name.
  7
? display counter                       COUNTER (in SORT-THE-BIDS) cannot be displayed when PROCESS-BIDS
 *ERROR - NO PROGRAM VARIABLE COUNTER   is the home program.
? step,1,procedure
 *S PROCEDURE AT PR.CALL-SORT-THE-BIDS
? step
 *S PROCEDURE AT P.SORT-TH_PR.SORTING   SORT-THE-BIDS is designated the home program when execution is
? set,breakpoint,pr.done-sorting        suspended within that program.
? go
 *B #2, AT PR.DONE-SORTING              NUMBER-OF-BIDS can be displayed, because it is defined in the
? d,#home                               Data Division of SORT-THE-BIDS.
 #HOME = P.SORT-TH
? display number-of-bids                COUNTER can be displayed when SORT-THE-BIDS is the home program.
  7
? display counter                       HIGH-BID cannot be displayed when SORT-THE-BIDS is the home
  8                                     program.
? display high-bid
 *ERROR - NO PROGRAM VARIABLE HIGH-BID  The SET,HOME command designates PROCESS BIDS as the HOME program.
? set,home,process                      Only the first seven characters of PROCESS-BIDS are specified.
? display high-bid
    <<HIGH BID                          HIGH-BID can be displayed when PROCESS-BIDS is the home program.
? display counter
 *ERROR - NO PROGRAM VARIABLE COUNTER   COUNTER cannot be displayed.
? go


        CUSTOMER      BID
        ID NUMBER

          CUST4      9111.32    <<HIGH BID

          CUST3      9062.21
          CUST2      5544.62
          CUST7      3344.22
          CUST8      3189.44
          CUST5      2266.44
          CUST9      0010.13

 *T #17, END IN L.125
? display counter
 *ERROR - NO PROGRAM VARIABLE COUNTER
? set,home,sort-th
? display counter
  8
? quit
 DEBUG TERMINATED
```

Figure 4-1. Home Program Example

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  PROCESS-BIDS.
*
*    THIS PROGRAM READS IN BIDS SUBMITTED FOR ONE ITEM
*    AT AN AUCTION AND SORTS THE BIDS IN DESCENDING
*    ORDER.  RESULTS ARE PRINTED OUT, AND THE HIGHEST
*    BID IS INDICATED.
*
*    SORTING IS DONE BY SUBPROGRAM SORT-THE-BIDS.
*
*    EACH INPUT LINE TAKES THE FORM:
*      CUSTOMER-ID                PICTURE X(5).
*      BID                        PICTURE 9999V99.
*
 ENVIRONMENT DIVISION.

 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  CYBER-170.
 OBJECT-COMPUTER.  CYBER-170.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
     SELECT IN-FILE ASSIGN TO "BIDS".
     SELECT OUT-FILE ASSIGN TO "OUTPUT".

 DATA DIVISION.

 FILE SECTION.

 FD IN-FILE
     LABEL RECORD IS OMITTED
     DATA RECORD IS LINE-IN.

 01  LINE-IN.
     05  CUSTOMER-ID             PICTURE X(5).
     05  BID                     PICTURE 9999V99.
     05  FILLER                  PICTURE X(9).

 FD OUT-FILE
     LABEL RECORD IS OMITTED
     DATA RECORD IS LINE-OUT.

 01  LINE-OUT.
     05  FILLER                  PICTURE X(10).
     05  CUSTOMER-ID             PICTURE X(10).
     05  BID                     PICTURE 9999.99.
     05  HIGH-BID-OR-SPACES      PICTURE X(13).


 WORKING-STORAGE SECTION.

 01  SPACE-LINE                  PICTURE X(100) VALUE SPACES.

 01  HEADING-1                   PICTURE X(25)
         VALUE "        CUSTOMER      BID".

 01  HEADING-2                   PICTURE X(20)
         VALUE "        ID NUMBER".

 01  HIGH-BID                    PICTURE X(13)
         VALUE "   <<HIGH BID".

 01  BID-INFORMATION.
     05  NUMBER-OF-BIDS          PICTURE 9V.
     05  BID-TABLE               OCCURS 10 TIMES
                                 INDEXED BY BID-INDEX.
         10  BID                 PICTURE 9999V99.
         10  CUSTOMER-ID         PICTURE X(5).

```

Figure 4-2. Main Program PROCESS-BIDS (Sheet 1 of 2)

```
PROCEDURE DIVISION.

INITIALIZATION SECTION.

OPEN-FILES.
    OPEN INPUT IN-FILE
         OUTPUT OUT-FILE.
INITIALIZE-VALUES.
    MOVE ZERO TO NUMBER-OF-BIDS.

PROCESS-A-BID SECTION.

READ-A-BID.
    READ IN-FILE AT END GO TO SORTING.
    ADD 1 TO NUMBER-OF-BIDS.
    MOVE CORRESPONDING LINE-IN
            TO BID-TABLE (NUMBER-OF-BIDS).
READ-NEXT-BID.
    GO TO READ-A-BID.

SORTING SECTION.

CALL-SORT-THE-BIDS.
    CALL "SORT-TH"
        USING BID-INFORMATION.

WRITE-RESULTS SECTION.
WRITE-HEADINGS.
    WRITE LINE-OUT FROM SPACE-LINE.
    WRITE LINE-OUT FROM SPACE-LINE.
    WRITE LINE-OUT FROM HEADING-1.
    WRITE LINE-OUT FROM HEADING-2.
WRITE-HIGH-BID.
    WRITE LINE-OUT FROM SPACE-LINE.
    MOVE HIGH-BID TO HIGH-BID-OR-SPACES OF LINE-OUT.
    MOVE CORRESPONDING BID-TABLE (1) TO LINE-OUT.
    WRITE LINE-OUT.
    WRITE LINE-OUT FROM SPACE-LINE.
WRITE-REMAINING-BIDS.
    PERFORM WRITE-A-BID VARYING BID-INDEX FROM 2 BY 1
        UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
    GO TO END-WRITE.
WRITE-A-BID.
    MOVE CORRESPONDING
          BID-TABLE (BID-INDEX) TO LINE-OUT.
    WRITE LINE-OUT.
END-WRITE.
    WRITE LINE-OUT FROM SPACE-LINE.

END-OF-RUN SECTION.

CLOSE-FILES.
    CLOSE IN-FILE, OUT-FILE.
STOP-RUN.
    STOP RUN.

```

Figure 4-2. Main Program PROCESS-BIDS (Sheet 2 of 2)

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  SORT-THE-BIDS.
*
*    THIS SUBPROGRAM SORTS THE BIDS FOR THE MAIN
*    PROGRAM PROCESS-BIDS.
*
 ENVIRONMENT DIVISION.

 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  CYBER-170.
 OBJECT-COMPUTER.  CYBER-170.

```

Figure 4-3. Subprogram SORT-THE-BIDS (Sheet 1 of 2)

```
INPUT-OUTPUT SECTION.

FILE-CONTROL.
    SELECT SORT-FILE ASSIGN TO SFILE.

DATA DIVISION.

FILE SECTION.

SD  SORT-FILE
    RECORD CONTAINS 11 CHARACTERS
    DATA RECORD IS SORT-RECORD.

01  SORT-RECORD.
    05  BID                     PICTURE 9999V99.
    05  CUSTOMER-ID             PICTURE X(5).

WORKING-STORAGE SECTION.

01  COUNTER                     PICTURE 99.

LINKAGE SECTION.

01  BID-INFORMATION.
    05  NUMBER-OF-BIDS          PICTURE 9V.
    05  BID-TABLE               OCCURS 10 TIMES
                                INDEXED BY BID-INDEX.
        10  BID                 PICTURE 9999V99.
        10  CUSTOMER-ID         PICTURE X(5).

PROCEDURE DIVISION USING BID-INFORMATION.

SORTING SECTION.

SORT-STATEMENT.
    SORT SORT-FILE
            ON DESCENDING KEY BID OF SORT-RECORD
          INPUT PROCEDURE IS SORT-IN-PROC
          OUTPUT PROCEDURE IS SORT-OUT-PROC.
DONE-SORTING.
    EXIT PROGRAM.

SORT-IN-PROC SECTION.

START-OF-SECTION.
    PERFORM SORTING-PARAGRAPH VARYING COUNTER FROM 1 BY 1
        UNTIL COUNTER IS GREATER THAN NUMBER-OF-BIDS.
    GO TO END-OF-SECTION.
SORTING-PARAGRAPH.
    MOVE CORRESPONDING BID-TABLE (COUNTER)
          TO SORT-RECORD.
    RELEASE SORT-RECORD.
END-OF-SECTION.

SORT-OUT-PROC SECTION.

START-OF-SECTION.
    PERFORM SORTING-PARAGRAPH VARYING COUNTER FROM 1 BY 1
        UNTIL COUNTER IS GREATER THAN NUMBER-OF-BIDS.
    GO TO END-OF-SECTION.
SORTING-PARAGRAPH.
      RETURN SORT-FILE RECORD
          AT END GO TO END-OF-SECTION.
      MOVE CORRESPONDING SORT-RECORD
              TO BID-TABLE (COUNTER).
END-OF-SECTION.
```

Figure 4-3. Subprogram SORT-THE-BIDS (Sheet 2 of 2)

```
CUST7334422
CUST2554462
CUST8318944
CUST5226644
CUST3906221
CUST4911132
CUST9001013
```

Figure 4-4. Input File BIDS

REFERENCING LOCATIONS OUTSIDE THE HOME PROGRAM

In many CID commands, you can specify line numbers and procedure-name references outside the home program by using program unit names and program name qualification. The commands of this sort described in this guide are as follows:

```
CLEAR,BREAKPOINT

CLEAR,TRAP

LIST,BREAKPOINT

LIST,MAP

LIST,TRAP

LIST,VALUES

SAVE,BREAKPOINT

SAVE,TRAP

SET,BREAKPOINT

SET,HOME

SET,TRAP

STEP
```

PROGRAM UNIT NAMES

Program unit names can be specified in many CID commands, including the SET,BREAKPOINT; SET,TRAP; and LIST,VALUES commands. Program unit names take the following form

```
P.program-unit
```

where program-unit specifies a main program or a subprogram. Only the first seven characters in the program unit name should be specified. If one or more of these characters are hyphens, the name must be enclosed in dollar signs ($). For example, the program unit PREPARE-PAYROLL is referenced as P.PREPARE, and the program unit COUNT-THE-BILLS is referenced as P.$COUNT-T$.

Program unit names used alone specify entire program units. You can specify entire program units when clearing and listing breakpoints; when setting, clearing, and listing traps; and when entering the STEP command.

You can also specify program unit names in the LIST,VALUES command

```
LIST,VALUES,program-list
```

where program-list is a list of program unit names in the form P.program-unit. List elements are separated by commas. When the LIST,VALUES command is entered in this way, values in the specified program units are listed.

Examples of CID commands that use program unit names to specify entire program units are as follows:

```
LIST,VALUES,P.$END-IT$,P.PROGRAM
```

This command lists the values in the program units END-IT and PROGRAM. END-IT is enclosed in dollar signs, because it contains a hyphen.

```
LV,P.$END-IT$,P.PROGRAM
```

This command has the same effect as the previous example.

```
CLEAR,BREAKPOINT,P.INVNTRY
```

This command clears all breakpoints in the program unit INVNTRY.

```
LIST,TRAP,P.$TRY-ONE$
```

This command lists all traps in the program unit TRY-ONE.

PROGRAM NAME QUALIFICATION

Program unit names can be combined with line number references and procedure-name references to form program name qualification. This notation allows you to specify single lines in program units outside the home program. Program name qualification is called qualification notation in other CID manuals and appears as a program unit name linked with an underline to a location:

```
P.program-unit_loc
```

In this notation, loc can be a line number reference or a procedure-name reference.

One example of program-name qualification is as follows:

```
P.PREPARE_L.215
```

This notation refers to line 215 in the program unit PREPARE.

Another example is as follows:

```
P.$COUNT-T$_PR.WRITE-A-LINE
```

This notation refers to the beginning of the WRITE-A-LINE section or paragraph in the program unit COUNT-T.

Program name qualification can be used anywhere line number references and procedure-name references are allowed. For example, you might use this notation to set a breakpoint, as follows:

```
SET,BREAKPOINT,P.$ADD-ONE$_PR.REPORT-ERROR
```

This command sets a breakpoint at the beginning of the REPORT-ERROR paragraph or section in the program unit ADD-ONE.

You can also use program name qualification in the ellipsis notation. An example of this usage is as follows:

```
SET,TRAP,LINE,P.PROG_L.110...P.PROG_L.127
```

This command sets a LINE trap with a scope of lines 110 through 127 in the program unit PROG.

CID uses the program name notation when it issues STEP command, trap, and breakpoint report messages. Whenever the home program changes as a result of execution being suspended, the new home program name is listed along with the line number. For example, if you enter the STEP command, the following message might be issued:

```
*S LINE AT P.ADD-ONE_L.137
```

This message indicates that the home program has changed to ADD-ONE. Execution is suspended at line 137 of the program unit ADD-ONE. If you enter the STEP command again, the following message might be issued:

```
*S LINE AT L.138
```

This message indicates that execution is suspended at line 138. The home program is still ADD-ONE.

DEBUGGING AIDS FOR MULTIPLE PROGRAM UNITS

CID provides features that can be helpful when you are debugging a main program with subprograms. The following paragraphs describe how you can set traps with scopes outside the home program, specify a scope for the STEP command, and enter a command to list load map information.

TRAP SCOPE PARAMETER

The trap scope can consist of an entire program unit or of several lines within a program unit. To specify the entire program unit, you use the program unit name. To specify several lines, you use program name qualification within the ellipsis notation.

Examples of setting traps with these kinds of scopes are as follows:

```
SET,TRAP,PROCEDURE,P.ACCOUNT
```

This command sets a PROCEDURE trap in the program unit ACCOUNT. Execution is suspended every time the beginning of a paragraph or section in the Procedure Division of ACCOUNT is reached.

```
SET,TRAP,LINE,P.PROG_L.135...P.PROG_L.187
```

This command sets a LINE trap in lines 135 through 187 in the program unit PROG. Execution is suspended whenever the beginning of one of these lines is reached.

STEP SCOPE PARAMETER

The STEP command has a scope parameter similar to the scope parameter in the SET,TRAP command. The format of a STEP command with a scope is as follows:

```
STEP,n,type,scope
```

In this command, the scope is either a range of lines specified by the ellipsis notation or a program unit name. The parameter n is the number of lines or procedure-names to count before resuming execution; type is either LINE or PROCEDURE.

When the scope is specified, only lines (or procedure-names) within that scope are counted, as shown in the following examples:

```
STEP,3,LINE,P.PAYROLL
```

This command causes execution to be resumed until three lines in the program unit PAYROLL have been counted. Lines in any other program unit are not counted.

```
STEP,5,PROCEDURE,P.PAY_L.85...P.PAY_L.132
```

This command causes execution to be resumed until five procedure-name lines in lines 85 through 132 of the program unit PAY have been counted. Procedure-name lines outside of lines 85 through 132 are not counted.

A debug session, in which the STEP command with a scope is entered, is shown in figure 4-5. This session uses the program units PROCESS-BIDS and SORT-THE-BIDS and the input file BIDS, all shown earlier in this section.

LIST,MAP COMMAND

The LIST,MAP command displays load map information. This command is useful when you are debugging multiple program units, because it provides a list of all modules currently in memory, including the names of every program unit being debugged. The LIST,MAP command can also provide information about specific modules. This information consists of the first word address (FWA), the length (in octal words), and the entry point names associated with each module.

```
CYBER INTERACTIVE DEBUG
? step,1,procedure,L.73...L.76
 *S PROCEDURE AT PR.INITIALIZATION
? step
 *S PROCEDURE AT PR.OPEN-FILES
? step

          CUSTOMER       BID
          ID NUMBER

             CUST4      9111.32    <<HIGH BID

             CUST3      9062.21
             CUST2      5544.62
             CUST7      3344.22
             CUST8      3189.44
             CUST5      2266.44
             CUST9      0010.13

 *T #17, END IN L.125
? quit
 DEBUG TERMINATED
```

The STEP command is entered with a scope of lines 73 through through 76. The INITIALIZATION section is in this scope.

The STEP command is repeated. The OPEN-FILES paragraph is within the scope.

The STEP command is repeated. Execution is not suspended until program termination, because no procedure-name lines within line 73 through 76 are encountered.

Figure 4-5. STEP Scope Example

The LIST,MAP command has the following forms:

LIST,MAP

This form lists the names of all modules (including program units) in the field length.

LIST,MAP,program-list

For each specified program unit, this form lists the name of the program unit, the first word address (FWA), the length (in octal words), and the entry point names. The program-list is a list of program units in the form P.program-unit. List elements are separated by commas.

The short form of LIST,MAP is LM. Examples of the LIST,MAP command are shown in figure 4-6. These examples use the program units PROCESS-BIDS and SORT-THE-BIDS and the input file BIDS, all shown earlier in this section.

SAMPLE DEBUG SESSIONS: MULTIPLE PROGRAM UNITS

The sample debug sessions shown in figures 4-7 and 4-8 show how CID processes multiple program units. Both debug sessions use the program units PROCESS-BIDS and SORT-THE-BIDS and the input file BIDS, all shown earlier in this section.

The following concepts are illustrated in the sample debug sessions:

Initially, the main program (PROCESS-BIDS) is the home program.

Program name qualification must be used to set breakpoints and traps outside the home program. Unqualified line numbers and procedure-names can be used for breakpoints and traps inside the home program.

CID informs you when the home program has changed by using program name qualification in STEP, breakpoint, and trap report messages.

COBOL programs

```
? list,map
 DBUG.,    PROCESS,   /CCOMMON/,   /C.HASHV/,   SORT-TH,   C$ADSUB
 C$COMIO,   C$EDIT,   C$ENTRY,   C$FILLT,   C$INIT,   /STP.END/,   C$MOVE
 C$MSG,   C$OMM,   C$SMEX,   C$SORT4,   C$SQOC,   C$SQRD,   C$SQWR
 C$STOPR,   C$UO8R1,   C$ADVAN,   C$CLOSF,   C$COLSQ,   C$CVCS,   C$LDCAP
 C$PRTRC,   C$RMASK,   C$STRP,   C$ZN,   C$R1U06,   SMSRTSZ,   SMCON7
 SMCIO,   SMLOAD,   CPU.CPM,   CPU.SYS,   CMF.AGR,   CMF.ALF,   CMF.FGR
 CMF.FRF,   CMF.GFS,   CMF.POE,   LOD=,   FDL.MMI,   /FDL.COM/,   UCLOAD
 CTL$LBL,   CTL$RM,   CTL$SKP,   CTL$WR,   CTRL$AA,   GPWR$RM,   CMM.CIA
 CMF.CSF,   CMF.DOE,   CMM.FFA,   CMM.GOA,   CMF.GOS,   CMM.R,   CMF.SLF
 FDL.RES,   ERR$RM,   LIST$RM,   RM$SYS=,   RM$X,   CPU.MVE,   CMM.MEM
 CMM.POA
? list,map,p.process
 PROGRAM - PROCESS,   FWA = 3447B,   LENGTH = 651B
 ENTRY POINTS - PROCESS
? list,map,p.$sort-th$
 PROGRAM - SORT-TH,   FWA = 4320B,   LENGTH = 435B
 ENTRY POINTS - SORT-TH
```

Figure 4-6. LIST,MAP Examples

```
 CYBER INTERACTIVE DEBUG
? d,#home
 #HOME = P.PROCESS
? step,1,procedure,p.$sort-th$
 *S PROCEDURE AT P.SORT-TH_PR.SORTING
? d,#home
 #HOME = P.SORT-TH
? set,trap,procedure,p.process_l.97...p.process_l.105
? go
 *T #1, PROCEDURE IN P.PROCESS_PR.WRITE-RESULTS
? go
 *T #1, PROCEDURE IN PR.WRITE-HEADINGS
? go


        CUSTOMER      BID
        ID NUMBER
 *T #1, PROCEDURE IN PR.WRITE-HIGH-BID
? go

          CUST4      9111.32    <<HIGH BID

          CUST3      9062.21
          CUST2      5544.62
          CUST7      3344.22
          CUST8      3189.44
          CUST5      2266.44
          CUST9      0010.13

 *T #17, END IN L.125
? quit
 DEBUG TERMINATED
```

Initially, the main program PROCESS-BIDS is home program.

The STEP command is entered with program unit SORT-THE-BIDS as scope. Execution is not suspended until a procedure-name line in SORT-THE-BIDS is encountered.

Program name qualification is used in the STEP report message to indicate that the home program has changed to SORT-THE-BIDS.

A PROCEDURE trap is set with lines 97 through 105 in the program unit PROCESS-BIDS as the scope.

Program name qualification is used in the trap report message to indicate the home program has changed to PROCESS-BIDS.

Program name qualification is not used, because the home program has not changed.

No more PROCEDURE traps occur, because execution is outside the trap scope.

Figure 4-7. Sample Debug Session A

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,pr.sorting
? set,breakpoint,p.$sort-th$_pr.sorting
? list,breakpoint,*
 *B #1 = PR.SORTING,   *B #2 = P.$SORT-TH$_PR.SORTING
? go
 *B #1, AT PR.SORTING
? d,#home
 #HOME = P.PROCESS
? go
 *B #2, AT P.$SORT-TH$_PR.SORTING
? d,#home
 #HOME = P.SORT-TH
? step,1,line,p.process
 *S LINE AT P.PROCESS_L.98
? step
 *S LINE AT L.99
? go

       CUSTOMER     BID
       ID NUMBER

         CUST4     9111.32    <<HIGH BID

         CUST3     9062.21
         CUST2     5544.62
         CUST7     3344.62
         CUST8     3189.44
         CUST5     2266.44
         CUST9     0010.13

 *T #17, END IN L.125
? quit
 DEBUG TERMINATED
```

Because no program name qualification was used, the breakpoint is set at the SORTING section in the home program PROCESS-BIDS.

Program name qualification is necessary when setting breakpoints outside the home program.

The home program is still PROCESS-BIDS; note that CID did not use program name qualification in the breakpoint report message.

CID uses program name qualification to indicate that the home program has changed to SORT-THE-BIDS.

A STEP command is entered with program unit PROCESS-BIDS as the scope. Execution is not suspended until a line in PROCESS-BIDS is reached.

Program name qualification is used in the STEP report message, because the home program has changed to PROCESS-BIDS.

No program name qualification is used, because the home program has not changed.

Figure 4-8. Sample Debug Session B

AUTOMATIC EXECUTION OF CID COMMANDS

In many cases, you will find it necessary to enter the same command or sequence of commands repeatedly during the course of a debug session. This type of situation is illustrated in figure 5-1. (This debug session uses the program FIND-HIGH-BID with input file BIDS, both shown in section 3.) In figure 5-1, the DISPLAY command is entered each time a breakpoint occurs. Entering the DISPLAY command in this example is a tedious process. It is also difficult to enter the lengthy command without making an error.

To eliminate the need for repeatedly entering sequences of commands, CYBER Interactive Debug (CID) allows you to define, save, and automatically execute command sequences. This feature can be used to improve debugging efficiency whenever the same group of CID commands must be entered repeatedly. Automatic command execution is commonly used when you are debugging parts of the program that execute many times (for example, as the result of a PERFORM statement).

COMMAND SEQUENCES

A command sequence, which is a series of CID commands, is executed automatically either when certain conditions occur or when you enter the appropriate command from the terminal.

There are three ways to 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 breakpoint or trap occurs. A sequence defined in this manner is called a breakpoint body or trap body.

By defining a command sequence called a group. A group is 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 are executed by a READ command, which is either entered at the terminal or executed by another command sequence.

During normal execution, CID prompts you for 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. You do not get control during sequence execution unless you provide for it using the PAUSE command, described later in this section.

Command sequences can be nested; that is, command sequences can be executed by other command sequences. However, a command sequence must have finished executing before it can be executed again. (It cannot execute itself, directly or indirectly.)

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.94
? go
 *B #1, AT L.94
? display number-of-bids
  1
? display bid-table (1)
 334422CUST7
? go
 *B #1, AT L.94
? display number-of-bids
  2
? display bid-table (2)
 554462CUST2
? go
 *B #1, AT L.94
? display number-of-bids
  3
? display bid-table (3)
 318944CUST8
? go
 *B #1, AT L.94
? display number-of-bids
  4
? display bid-table (4)
 226644CUST5
? go
 *B #1, AT L.94
? display number-of-bids
  5
? display bid-table (5)
 906221CUST3
? go
 *B #1, AT L.94
? display number-of-bids
  6
? display bid-table (6)
 911132CUST4
? go
 *B #1, AT L.94
? display number-of-bids
  7
? display bid-table (7)
 001013CUST9
? go


         CUSTOMER     BID
         ID NUMBER

           CUST4      9111.32    <<HIGH BID

           CUST3      9062.21
           CUST2      5544.62
           CUST7      3344.22
           CUST8      3189.44
           CUST5      2266.44
           CUST9      0010.13

 *T #17, END IN L.160
? quit
 DEBUG TERMINATED
```

Figure 5-1. Commands Entered Repeatedly

COLLECT MODE

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

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

MULTIPLE COMMAND ENTRY

You can enter more than one command on a single line by separating the commands with a semicolon, as in the following example:

```
DISPLAY X;SET,BREAKPOINT,L.134;GO
```

In interactive mode, CID does not execute the commands until you press the RETURN key; it then executes the commands in the order you entered them. In collect mode, the commands are not executed immediately, but are included in the command sequence for later execution.

SEQUENCE COMMANDS

CID provides a set of commands intended specifically for use with command sequences. The commands of this type described in this user's guide are summarized in table 5-1.

TABLE 5-1. SEQUENCE COMMANDS

Command	Description
EXECUTE	Resumes execution of your program
GO	Resumes the process most recently suspended
PAUSE	Temporarily suspends execution of the current command sequence and reinstates interactive mode allowing commands to be entered from the the terminal
READ	Initiates execution of a command sequence defined as a group or stored on a file; reestablishes breakpoint, trap, and group definitions stored on a file

BREAKPOINTS AND TRAPS WITH BODIES

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

```
IN COLLECT MODE
?
```

You then enter the commands that make up the body. Each command entered while CID is in collect mode becomes part of the body. CID scans the command for syntax 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 question mark (?) prompt or at the end of a command line. CID then displays:

```
END COLLECT MODE
?
```

You can then continue the session.

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

```
SET,BREAKPOINT,L.8 [
MOVE 5 TO NUMBER-OF-BIDS
SET BID-INDEX UP BY 1
DISPLAY NUMBER-OF-BIDS, BID-INDEX
]
```

When a breakpoint or trap 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; CID does not give control to you upon completion of the sequence.

A short breakpoint or trap body can be specified in the line containing the SET,BREAKPOINT or SET,TRAP command. For example, the following SET,BREAKPOINT command causes #LINE and data item A to be displayed when line 507 is reached:

```
SET,BREAKPOINT,L.507 [D,#LINE;DISPLAY A]
```

This breakpoint body can also be defined as follows:

```
SET,BREAKPOINT,L.507 [
D,#LINE
DISPLAY A
]
```

When a breakpoint or trap 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 breakpoint or trap body by including a DISPLAY command in the sequence.

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.

Figure 5-2 shows a debug session in which two breakpoint bodies are defined. The session in figure 5-2 is a variation of the session in figure 5-1; both sessions display the same data and use the program FIND-HIGH-BID with input file BIDS (shown in section 3).

Figure 5-3 shows how a trap body can be used to trace program execution. In the session, a PROCEDURE trap with a body is set. The trap body displays the value of the debug variable #PROC. Similarly, you can set a LINE trap that displays #LINE to trace execution line-by-line. The debug session in figure 5-3 uses the program FIND-HIGH-BID with input file BIDS.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,l.94 [
 IN COLLECT MODE
? display "number-of-bids:  ", number-of-bids
? display "bid-table (number-of-bids):  ", bid-table (number-of-bids)  <---- Breakpoint body
? display " "
? ]
 END COLLECT
? go
 NUMBER-OF-BIDS:   1
 BID-TABLE (NUMBER-OF-BIDS):  334422CUST7

 NUMBER-OF-BIDS:   2
 BID-TABLE (NUMBER-OF-BIDS):  554462CUST2

 NUMBER-OF-BIDS:   3
 BID-TABLE (NUMBER-OF-BIDS):  318944CUST8

 NUMBER-OF-BIDS:   4
   BID-TABLE (NUMBER-OF-BIDS):  226644CUST5    <------------------ Breakpoint body output

   NUMBER-OF-BIDS:   5
   BID-TABLE (NUMBER-OF-BIDS):  906221CUST3

   NUMBER-OF-BIDS:   6
   BID-TABLE (NUMBER-OF-BIDS):  911132CUST4

   NUMBER-OF-BIDS:   7
   BID-TABLE (NUMBER-OF-BIDS):  001013CUST9


           CUSTOMER     BID
           ID NUMBER

             CUST4     9111.32    <<HIGH BID

             CUST3     9062.21
             CUST2     5544.62
             CUST7     3344.22
             CUST8     3189.44
             CUST5     2266.44
             CUST9     0010.13

   *T #17, END IN L.160
  ? quit
   DEBUG TERMINATED
```

Figure 5-2. Debug Session With Breakpoint Body

```
CYBER INTERACTIVE DEBUG
? set,trap,procedure,* [d,#proc]
? go
 #PROC = P.FIND-HI_PR.INITIALIZATION
 #PROC = P.FIND-HI_PR.OPEN-FILES
 #PROC = P.FIND-HI_PR.INITIALIZE-VALUES
 #PROC = P.FIND-HI_PR.PROCESS-A-BID
 #PROC = P.FIND-HI_PR.READ-A-BID
 #PROC = P.FIND-HI_PR.READ-NEXT-BID
 #PROC = P.FIND-HI_PR.READ-A-BID
 #PROC = P.FIND-HI_PR.READ-NEXT-BID
 #PROC = P.FIND-HI_PR.READ-A-BID
 #PROC = P.FIND-HI_PR.READ-NEXT-BID
 #PROC = P.FIND-HI_PR.READ-A-BID
     .
     .
     .
```

Figure 5-3. Tracing Program Execution

DISPLAYING BREAKPOINT AND TRAP BODIES

You can display a list of the commands in a breakpoint body by specifying the breakpoint location in the following LIST,BREAKPOINT commands:

LIST,BREAKPOINT,loc-list

> This command displays the complete definitions, including the bodies, of the breakpoints at statements given by locations in the loc-list. Each location has one of the forms L.n, PR.name, P.program-unit_L.n, or P.program-unit_PR.name.

LIST,BREAKPOINT,number-list

> This command displays the complete definitions, including the bodies (if any), of the breakpoints with the listed numbers. Each breakpoint number has the form #n. (CID assigns the number when the breakpoint is established.)

Other forms of the LIST,BREAKPOINT command list the breakpoint location but not the commands in the body.

To display a list of the commands in a trap body, enter the following form of the LIST,TRAP command:

LIST,TRAP,number-list

> This command displays the types, locations, and bodies (if any) of the traps with the listed numbers. Each trap number has the form #n. (CID assigns the trap numbers when the traps are established.)

Other forms of the LIST,TRAP command list only the trap type and location.

Figure 5-4 illustrates a LIST,BREAKPOINT command for the breakpoint established in figure 5-2.

GROUPS

A group is a sequence of commands established and assigned a name during a debug session, but not explicitly associated with a breakpoint or trap. A group exists until you clear it or terminate the debug session and is 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 breakpoint and trap bodies. Any number of CID commands subsequently entered become part of the sequence until you terminate the sequence by entering a right bracket. The short form of SET,GROUP is SG.

The command to execute a group is

READ,name

where name is the group name assigned in the SET,GROUP command. You can issue a READ command directly from the terminal 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 returns to CID (if the READ was entered from the terminal) or to the next command in the sequence that issued the READ.

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

```
SET,GROUP,GRPA [
SET INDEX-A UP BY 1
DISPLAY "INDEX-A = ", INDEX-A
]
```

```
? list,breakpoint,#1
 *B #1 = L.94
 SET,BREAKPOINT,L.94 [
 DISPLAY "NUMBER-OF-BIDS:  ", NUMBER-OF-BIDS
 DISPLAY "BID-TABLE (NUMBER-OF-BIDS):  ", BID-TABLE (NUMBER-OF-BIDS)
 DISPLAY " "
 ]
?
```

Figure 5-4. Displaying a Breakpoint Body

This command sequence is executed by entering 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,GROUP, CLEAR,GROUP, and SAVE,GROUP commands.

You can list group information by entering the following commands:

LIST,GROUP,*

> This command lists the names and numbers of all groups defined for the current debug session; it does not list the commands contained in the groups.

LIST,GROUP,name-list

> This command lists the commands contained in the specified groups. Group names are separated by commas.

LIST,GROUP,number-list

> This command lists the commands contained in the groups identified by the listed numbers. Each group number has the form #n. (CID assigns the group numbers when the groups are established.)

Note that the first command form lists only the names and numbers of groups, whereas the second and third forms list the commands that make up the specified groups. The short form of LIST,GROUP is LG.

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

CLEAR,GROUP,*

> This command removes all currently-defined groups.

CLEAR,GROUP,name-list

> This command removes the specified groups.

CLEAR,GROUP,number-list

> This command removes the groups identified by the listed numbers. Each group number has the form #n.

The short form of CLEAR,GROUP is CG.

Figures 5-5 and 5-6 illustrate debug sessions using groups; both sessions use the program FIND-HIGH-BID with input file BIDS (shown in section 3). In figure 5-5, two breakpoints are set. When either breakpoint is reached, the READ command is issued from the terminal. In figure 5-6, the same breakpoints are established, except that a body containing a READ command is defined for each breakpoint. This causes the body to be executed automatically when the breakpoints are encountered. 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 5-6, 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.

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 syntax errors. If a syntax error is detected, CID displays an error message followed by a question mark (?), after which you can reenter the command. Other errors, however, such as nonexistent line number or data 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, displays the command in error, and issues a message followed by an input prompt (? for error messages; OK? for warning messages) on the next line. You then can instruct CID to disregard the command, replace the command with another command, or, in the case of warning messages, execute the command. The ways in which you can respond to error and warning messages are summarized in table 5-2.

TABLE 5-2. RESPONSES TO ERROR AND WARNING MESSAGES

Your Response	CID 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 (or sequence of commands separated by semicolons)	Execute the specified command line in place of the current command, and resume execution of the sequence.

```
 CYBER INTERACTIVE DEBUG
? set,group,dsply [
 IN COLLECT MODE
? display " "
? display "bid-table displayed by group:"
? display " "
? display bid-table (1)
? display bid-table (2)
? display bid-table (3)                     <---- A group is defined.
? display bid-table (4)
? display bid-table (5)
? display bid-table (6)
? display bid-table (7)
? ]
 END COLLECT
? set,breakpoint,l.97
? set,breakpoint,l.132
? go
 *B #1, AT L.97
? read,dsply <------------------------------ The READ command causes the group to execute.

 BID-TABLE DISPLAYED BY GROUP

 334422CUST7
 554462CUST2
 318944CUST8
 226644CUST5
 906221CUST3
 911132CUST4
 001013CUST9
? go
 *B #2, AT L.132
? read,dsply

 BID-TABLE DISPLAYED BY GROUP

 911132CUST4
 906221CUST3
 554462CUST2
 334422CUST7
 318944CUST8
 226644CUST5
 001013CUST9
? go


        CUSTOMER      BID
        ID NUMBER

          CUST4     9111.32   <<HIGH BID

          CUST3     9062.21
          CUST2     5544.62
          CUST7     3344.22
          CUST8     3189.44
          CUST5     2266.44
          CUST9     0010.13

 *T #17, END IN L.160
? quit
 DEBUG TERMINATED
```

Figure 5-5. Defining and Executing a Group

```
 CYBER INTERACTIVE DEBUG
? set,group,dsply [
 IN COLLECT MODE
? display " "
? display "bid-table displayed by group:"
? display " "
? display bid-table (1)
? display bid-table (2)
? display bid-table (3)                 <-- Group definition.
? display bid-table (4)
? display bid-table (5)
? display bid-table (6)
? display bid-table (7)
? ]
 END COLLECT
? set,breakpoint,l.97 [read,dsply]  }<--- Breakpoints are defined to read the group automatically.
? set,breakpoint,l.132 [read,dsply] }
? go

 BID-TABLE DISPLAYED BY GROUP

 334422CUST7
 554462CUST2
 318944CUST8                 <--------- Output caused by breakpoint at line 97.
 226644CUST5
 906221CUST3
 911132CUST4
 001013CUST9

 BID-TABLE DISPLAYED BY GROUP

 911132CUST4
 906221CUST3
 554462CUST2                 <--------- Output caused by breakpoint at line 132.
 334422CUST7
 318944CUST8
 226644CUST5
 001013CUST9


        CUSTOMER      BID
        ID NUMBER

          CUST4      9111.32    <<HIGH BID

          CUST3      9062.21
          CUST2      5544.62
          CUST7      3344.22
          CUST8      3189.44
          CUST5      2266.44
          CUST9      0010.13

 *T #17, END IN L.160
? quit
 DEBUG TERMINATED
```

Figure 5-6. Group Automatically Called From Breakpoint Bodies

An example of error processing during sequence execution is illustrated in figure 5-7. During execution of group ABC, CID issues warning and error messages. After each message is issued, CID gives you control. In response to the first message, NO is entered, and CID disregards the command. In response to the second message, a new command is entered and executed in place of the sequence command. In response to the third message, NO,SEQ is entered, instructing CID to disregard the incorrect command and all remaining commands in the sequence and to give you control.

RECEIVING CONTROL DURING SEQUENCE EXECUTION

Normally, a command sequence executes to completion without giving you control. 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 by using the PAUSE command.

PAUSE COMMAND

The purpose of the PAUSE command is to suspend execution of a command sequence. 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 when the PAUSE command is executed.

The PAUSE command is valid only in a command sequence; it cannot be entered directly from the terminal.

When a PAUSE command is encountered in a breakpoint or trap body, CID displays the breakpoint or trap 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

The functions of the GO and EXECUTE commands are identical except when the commands are executed following suspension of a command sequence. When program execution has been suspended by a breakpoint or trap, both commands resume program execution. However, when execution of a command sequence has been suspended, the GO and EXECUTE commands differ as follows:

GO resumes execution of the suspended sequence.

EXECUTE causes an immediate exit from the sequence and resumes execution of the program. The short form of EXECUTE is EXEC.

The debug session in figure 5-8 illustrates the PAUSE, GO, and EXECUTE commands. This session was produced using the program FIND-HIGH-BID and input file BIDS. The group LASTBID is defined with a PAUSE command in it. LASTBID is executed twice during the session. The first time the PAUSE command is encountered, the GO command is entered to resume execution of the command sequence. The second time the PAUSE command is encountered, the EXECUTE command is entered to resume execution of the program. Note that the DISPLAY command at the end of the sequence is not executed in this case.

```
 CYBER INTERACTIVE DEBUG
? set,group,abc [
 IN COLLECT MODE
? display bid-table (number-of-bids)
? move 899.23 to bid-table (1)
? move 8 to count
? display "end of group abc"
? ]
 END COLLECT
? set,breakpoint,pr.read-a-bid
? go
 *B #1, AT PR.READ-A-BID                                   Responding with NO causes the
? read,abc                                                 command in error to be disre-
 *CMD - ( DISPLAY BID-TABLE (NUMBER-OF-BIDS) )  *ERROR - SUBSCRIPT OUT   garded. Processing continues
 OF RANGE                                                  with the next command in the
? no <---------------------------------------------------  sequence.
 *CMD - ( MOVE 899.23 TO BID-TABLE (1) )  *WARN - --NO EDITING WILL BE
 DONE                                                      Responding with a CID command
 OK ? move 899.23 to bid of bid-table (1)<-----------------  causes the sequence command to
 *CMD - ( MOVE 8 TO COUNT )  *ERROR - NO PROGRAM VARIABLE COUNT   be replaced with the response
? no,seq <------------------------------------------------  command.
? go
 *B #1, AT PR.READ-A-BID                                   Responding with NO,SEQ causes
   .                                                       all of the commands remaining
   .                                                       in the sequence to be
   .                                                       disregarded.
```

Figure 5-7. Command Sequence Error Processing

```
 CYBER INTERACTIVE DEBUG
? set,group,lastbid [
 IN COLLECT MODE
? display "bid-number:  ", number-of-bids
? display "customer-id:  ",  customer-id of bid-table (number-of-bids)
? display "amount of bid:  ", bid of bid-table (number-of-bids)
? pause, "changes?" <-------------------------------- The group LASTBID contains a
? display "end of group"                               PAUSE command.
? ]
 END COLLECT
? set,breakpoint,l.94
? go
 *B #1, AT L.94
? read,lastbid
 BID-NUMBER:   1                                       The PAUSE command suspends
 CUSTOMER-ID:  CUST7                                   execution of the sequence,
 AMOUNT OF BID:   3344.22                              displays a message, and gives
 CHANGES? <-------------------------------------------  you control.
? move "ywqxt" to customer-id of bid-table (1)
? go <----------------------------------------------- The GO command causes execu-
 END OF GROUP                                          tion of the group to continue.
? go                                                   Note that END OF GROUP is
 *B #1, AT L.94                                        displayed.
? read,lastbid
 BID-NUMBER:   2
 CUSTOMER-ID:  CUST2
 AMOUNT OF BID:   5544.62                              The PAUSE command suspends
 CHANGES? <-------------------------------------------  execution.
? move 0 to bid of bid-table (2)
? execute <------------------------------------------ The EXECUTE command causes
 *B #1, AT L.94                                        program execution to resume.
? clear,breakpoint,*                                   END OF GROUP is not displayed.
? go


        CUSTOMER      BID
        ID NUMBER

          CUST4      9111.32    <<HIGH BID

          CUST3      9062.21
          YWQXT      3344.22
          CUST8      3189.44
          CUST5      2266.44
          CUST9      0010.13
          CUST2      0000.00

 *T #17, END IN L.160
? quit
 DEBUG TERMINATED
```

Figure 5-8. PAUSE Command Example

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 Breakpoint, Trap, 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 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,file-name

where file-name is the name of the file that contains the commands. CID reads the file and automatically executes the commands in the same manner as for a group. When execution of the commands is complete, program execution remains suspended, and control returns to you. To resume program execution, enter GO.

Executing commands from a file can be time-consuming since the file must be read each time the command 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, a file containing the following commands could be created with a text editor:

```
    DISPLAY NUMBER-OF-BIDS
    DISPLAY BID-TABLE (NUMBER-OF-BIDS)
```

If the file is called COMF, the command READ,COMF must be issued whenever the sequence is to be executed. An alternative is to create COMF as follows:

```
    SET,GROUP,GRPX [
    DISPLAY NUMBER-OF-BIDS
    DISPLAY BID-TABLE (NUMBER-OF-BIDS)
    ]
```

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 text editors under NOS and NOS/BE to create and edit files containing CID commands is described under Editing a Command Sequence.

SAVING BREAKPOINT, TRAP, 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 and breakpoint, trap, and group definitions on a separate file. You can print this file or make it permanent. There are two reasons for copying CID definitions to a file:

To preserve a copy of the definitions for use in current or subsequent debug sessions

To make it easier to edit command sequences with the system text editor

The following commands save CID definitions:

SAVE,BREAKPOINT,file-name,list

This command copies the definitions of the breakpoints specified in list to the named file; list is an optional list of breakpoint locations (L.n or PR.n) or breakpoint numbers (#n) separated by commas. If list is * or omitted, all breakpoints are saved. The short form of SAVE,BREAKPOINT is SAVEB.

SAVE,TRAP,file-name,type,scope

This command copies to the named file 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; if they are * or omitted, all existing traps are saved. The short form of SAVE,TRAP is SAVET.

SAVE,GROUP,file-name,list

This command copies the groups specified in list to the named file; list is an optional list of group names or numbers (#n) separated by commas. If list is * or omitted, all groups defined for the current session are saved. The short form of SAVE,GROUP is SAVEG.

The SAVE commands copy the complete definition of the specified breakpoints, traps, or groups to the specified file. (The definition of a breakpoint, trap, or group includes the SET command and any other commands in the body.)

You can combine breakpoint, trap, and group definitions on a single file by specifying the same file name for multiple SAVE commands. A single READ command reestablishes all the definitions stored in the file. Another way to combine definitions on a single file is to enter the command:

SAVE,*,file-name

This command copies all existing breakpoint, trap, and group definitions to the specified file.

Some examples of SAVE commands are as follows:

SAVE,BREAKPOINT,SBPF,*

This command copies to file SBPF all existing breakpoints.

SAVE,BREAKPOINT,BPFILE,L.10,P.SUBX_L.20

This command copies to BPFILE the definitions of the breakpoints established at line 10 of the home program and line 20 of program unit SUBX.

SAVE,BREAKPOINT,FILEA,#2,#5

This command copies to FILEA the definitions of breakpoints #2 and #5.

SAVEB,FILEA,#2,#5

This command has the same effect as the previous example.

SAVE,TRAP,TFILE,*

This command copies to TFILE all existing traps.

SAVE,TRAP,TTT,LINE,P.PROGA

This command copies to TTT the definition of all LINE traps established in program unit PROGA.

SAVET,TTT,LINE,P.PROGA

This command has the same effect as the previous example.

SAVE,GROUP,GFIL,WRT,RDD,GRPX

This command copies to GFIL the definitions of the groups named WRT, RDD, and GRPX.

SAVEG,GFIL,WRT,RDD,GRPX

This command has the same effect as the previous example.

The file on which the definitions are saved is a local file. If you want to access these definitions after logging out, you must make the file permanent.

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,file-name
```

where the named file contains the definitions. You can issue a READ command in the current session or in a later session. If READ,file-name is executed in the current session and the definitions previously saved on the file have not been removed by the appropriate CLEAR command, CID displays a message of the form:

```
*WARN - 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.

Note that the READ command only restores the definitions stored in the specified file; it does not cause the commands in the definitions to be executed.

The following example READ commands assume that GFIL and TTT are as defined in the preceding examples:

READ,TTT

This command restores the LINE trap definition contained in file TTT.

READ,GFIL

This command restores the group definitions contained in file GFIL.

Debug sessions using the SAVE,BREAKPOINT and READ commands are shown in figure 5-9. These debug sessions use the program FIND-HIGH-BID and input file BIDS, both shown in section 3. In the first session, two breakpoints and one trap are defined, all with bodies. The program is executed, and at the end of the session the breakpoints are saved on the local file BKFILE. In the second debug session, the breakpoint definitions are read from BKFILE using the READ command.

EDITING A COMMAND SEQUENCE

If you wish to make a change to a command sequence in a breakpoint body, trap body, or group, you can remove the definition with the appropriate CLEAR command and reenter the entire sequence. This procedure is both time-consuming and difficult for lengthy sequences.

CID provides two alternate methods for making changes to a command sequence:

You can save the breakpoint, trap, or group definition on a separate file and edit the file.

You can turn on veto mode and edit the sequence interactively each time the sequence is executed. See the CYBER Interactive Debug reference manual for an explanation of this method.

To apply the first method, you must suspend the current debug session.

SUSPENDING A DEBUG SESSION

CID provides the capability of suspending the current session, returning to system command mode, and 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 following commands suspend the current debug session, copy information about the session environment to a file, and return control to the operating system:

SUSPEND

This command saves the debug session on the local file ZZZZZDS.

SUSPEND,file-name

This command saves the debug session on the local file specified by file-name.

The information saved on the local file includes copies of:

The executing program

All trap, breakpoint, and group definitions

All CID internal tables

In short, the file contains all the information necessary to continue the session. (Program data files are not saved, but them are unaffected if you do not change their positions or log out.)

The information contained in the file created by a SUSPEND command is intended for use by CID only; you should not access this information directly. The suspend file preserves the status of a debug session exactly as it existed when the SUSPEND command was executed. The suspend file is a local file.

```
 CYBER INTERACTIVE DEBUG
? set,breakpoint,pr.sorting [
 IN COLLECT MODE
? d,#proc
? display "number of bids:  ", number-of-bids
? ]
 END COLLECT
? set,breakpoint,l.132 [
 IN COLLECT MODE
? d,#proc
? pause
? ]
 END COLLECT
? set,trap,procedure,l.77...l.87 [d,#proc]
? go
 #PROC = P.FIND-HI_PR.INITIALIZATION
 #PROC = P.FIND-HI_PR.OPEN-FILES
 #PROC = P.FIND-HI_PR.INITIALIZE-VALUES
 #PROC = P.FIND-HI_PR.SORTING
 NUMBER-OF-BIDS:  7
 #PROC = P.FIND-HI_PR.WRITE-RESULTS
 *B #2, AT L.132
? move 2 to number-of-bids
? go


        CUSTOMER      BID
        ID NUMBER

          CUST4     9111.32   <<HIGH BID

          CUST3     9062.21

 *T #17, END IN L.160
? list,breakpoint,*
 *B #1 = PR.SORTING ,   *B #2 = L.132
? list,trap,*
 T #1 = PROCEDURE L.77...L.87
? save,breakpoint,bkfile,*◄──────────────── All breakpoints are saved on the file BKFILE.
? quit
 DEBUG TERMINATED
/lgo
 CYBER INTERACTIVE DEBUG
? list,breakpoint,* ◄────────────────────── No breakpoints exist at the beginning of the second
 NO BREAKPOINTS                             debug session.
? list,trap,*
 NO TRAPS
? read,bkfile ◄──────────────────────────── Breakpoint definitions are read from BKFILE.
? list,breakpoint,*
 *B #1 = PR.SORTING ,   *B #2 = L.132
? list,breakpoint,#1,#2
 *B #1 = PR.SORTING
 SET,BREAKPOINT,PR.SORTING [
 D,#PROC
 DISPLAY "NUMBER-OF-BIDS:  ", NUMBER-OF-BIDS
 ]
 *B #2 = L.132
 SET,BREAKPOINT,L.132 [
 D,#PROC
 PAUSE
 ]
? list,trap,*
 NO TRAPS
? quit
 DEBUG TERMINATED
```

Figure 5-9. Saving and Reading Command Sequences

In general, you should not log out after suspending the debug session, because the file positions of your input files are not saved. However, if your program has not begun reading from the input files or if it has finished reading from them, you can make the suspend file permanent and then log out. The suspended debug session can then be resumed in another terminal session.

You should not alter the status of any files used by your program after you issue a SUSPEND command. If you perform any file manipulation operations, such as REWIND, on files used by your program, you might not be able to restart the session normally.

To resume the suspended debug session, enter one of the following commands:

DEBUG(RESUME)

This command resumes the debug session that was suspended on the file ZZZZZDS.

DEBUG(RESUME,file-name)

This command resumes the debug session that was suspended on the file specified by file-name.

The session is then restored to its status as it existed at the time of suspension. All breakpoint, trap, and group definitions are restored, and all program and debug variables have the values that existed when SUSPEND was executed.

Remember that the most effective debug sessions are short and simple. Thus, it will rarely be necessary to use the SUSPEND/RESUME capability, except to edit command sequences.

EDITING PROCEDURE

To edit a breakpoint body, trap body, or command group, proceed as follows:

1. Save the breakpoint, trap, or group definition with the appropriate SAVE command.
2. Suspend the current session with the SUSPEND command.
3. Use a text editor to make desired changes to the command sequence.
4. Resume the session with the DEBUG(RESUME) command.
5. Remove the old breakpoint, trap, or group definition with the appropriate CLEAR command.
6. Establish the altered definition with the READ command.

After a SUSPEND, be sure that you do not modify or change the position of any files used by your program, because 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 (figure 5-10) and NOS/BE (figure 5-11). In the session, the group DISGRP is mistakenly defined to display BID-TALE (3). The command sequence is then edited to change the characters TALE to the characters TABLE.

INTERRUPTING AN EXECUTING SEQUENCE

A terminal interrupt allows you to gain control at any time during a debug session. If your program is executing at the time of the interrupt, the INTERRUPT trap occurs as described in section 3. However, if a command sequence is executing at the time of the interrupt, execution of the sequence is suspended and CID displays the message:

```
INTERRUPTED
?
```

The ways in which you can respond to this message are shown in table 5-3.

If CID is in the process of displaying information when the interrupt occurs, the information remaining to be printed is lost. A terminal interrupt is therefore an effective means of stopping excessive CID output.

TABLE 5-3. INTERRUPT RESPONSES

Your Response	CID Action
OK or YES	Resumes sequence execution at the point of the interrupt.
GO or NO,SEQ	Disregards all remaining commands in the sequence and resumes execution of the program.
Any CID command	Executes the specified command and resumes execution of the sequence at the point of the interrupt.

```
 CYBER INTERACTIVE DEBUG
? set,group,disgrp [                                    Group DISGRP is defined.
 IN COLLECT MODE
? display bid-table (1)
? display bid-table (2)
? display bid-tale (3)
? display bid-table (4)
? display bid-table (5)
? display bid-table (6)
? display bid-table (7)
? ]
 END COLLECT
? set,breakpoint,pr.sorting                             An error exists in the group.
? go                                                    The group must be edited.
 *B #1, AT PR.SORTING
? read,disgrp
 334422CUST7
 554462CUST2
 *CMD - ( DISPLAY BID-TALE (3) )  *ERROR - NO PROGRAM VARIABLE BID-TALE
? no,seq                                                The group is saved on file
? save,group,savfile,disgrp                             SAVFILE.
? suspend
 DEBUG SUSPENDED                                        The current session is
/xedit,savfile                                          suspended.
 XEDIT 3.1.00
?? locate/tale/                                         The group, saved on SAVFILE, is
DISPLAY BID-TALE (3)                                    edited with a text editor.
?? change/tale/table/
DISPLAY BID-TABLE (3)
?? quit
SAVFILE IS A LOCAL FILE
/debug(resume)                                          The debug session is resumed.
 CYBER INTERACTIVE DEBUG RESUMED
? clear,group,disgrp                                    The old group definition is
? read,savfile                                          removed.
? list,group,disgrp
 *G #1 = DISGRP                                         The corrected group definition
 SET,GROUP,DISGRP [                                     is read from SAVFILE.
 DISPLAY BID-TABLE (1)
 DISPLAY BID-TABLE (2)
 DISPLAY BID-TABLE (3)
 DISPLAY BID-TABLE (4)
 DISPLAY BID-TABLE (5)
 DISPLAY BID-TABLE (6)
 DISPLAY BID-TABLE (7)
 ]
? read,disgrp
 334422CUST7
 554462CUST2
 318944CUST8
 226644CUST5
 906221CUST3
 911132CUST4
 001013CUST9
? d,#proc
 #PROC = P.FIND-HI_PR.SORTING
? go

        CUSTOMER      BID
        ID NUMBER

          CUST4     9111.32    <<HIGH BID

          CUST3     9062.21
          CUST2     5544.62
          CUST7     3344.22
          CUST8     3189.44
          CUST5     2266.44
          CUST9     0010.13

 *T #17, END IN L.160
? quit
```

Figure 5-10. Editing a Command Sequence on NOS

```
CYBER INTERACTIVE DEBUG
?set,group,disgrp                                                    Group DISGRP is defined.
IN COLLECT MODE
?display bid-table (1)
?display bid-table (2)
?display bid-tale (3)
?display bid-table (4)
?display bid-table (5)
?display bid-table (6)
?display bid-table (7)
?]
END COLLECT
?set,breakpoint,pr.sorting                                           An error exists in the group.
?go                                                                  The group must be edited.
*B #1, AT PR.SORTING
?read,disgrp
334422CUST7
554462CUST2
*CMD - ( DISPLAY BID-TALE (3) )  *ERROR - NO PROGRAM VARIABLE BID-TALE
?no,seq                                                              The group is saved on file
?save,group,savfile,disgrp                                           SAVFILE.
?suspend
  DEBUG SUSPENDED                                                    The current session is
COMMAND- editor                                                      suspended.
..edit,savfile,seq
../tale/=/table/                                                     The group, saved on SAVFILE,
         1 CHANGE(S)                                                 is edited with a text editor.
..save,newfile,noseq                                                 The corrected group is saved
..bye                                                                on NEWFILE.
COMMAND- debug(resume)
CYBER INTERACTIVE DEBUG RESUMED                                      The debug session is resumed.
?clear,group,disgrp
?read,newfile                                                        The old group definition is
?list,group,disgrp                                                   removed.
*G #1 = DISGRP
SET,GROUP,DISGRP [                                                   The corrected group definition
DISPLAY BID-TABLE (1)                                                is read from NEWFILE.
DISPLAY BID-TABLE (2)
DISPLAY BID-TABLE (3)
DISPLAY BID-TABLE (4)
DISPLAY BID-TABLE (5)
DISPLAY BID-TABLE (6)
DISPLAY BID-TABLE (7)
]
?read,disgrp
334422CUST7
554462CUST2
318944CUST8
226644CUST5
906221CUST3
911132CUST4
001013CUST9
?d,#proc
 #PROC = P.FIND-HI_PR.SORTING
?go

        CUSTOMER      BID
        ID NUMBER

          CUST4     9111.32   <<HIGH BID

          CUST3     9062.21
          CUST2     5544.62
          CUST7     3344.22
          CUST8     3189.44
          CUST5     2266.44
          CUST9     0010.13

 *T #17, END IN L.160
?quit
```

Figure 5-11. Editing a Command Sequence on NOS/BE

STANDARD CHARACTER SETS A

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 table A-1 are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

TABLE A-1. STANDARD CHARACTER SETS

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	52	!	12-8-7	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	72	<	12-8-4	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).

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 CYBER Interactive Debug (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 the commands you enter are not executed, but are included in a group, trap, or breakpoint body. Collect mode is initiated by a left bracket ([) at the end of a SET,TRAP, SET,GROUP, or SET,BREAKPOINT command; collect mode is terminated by a right bracket (]).

Debug Mode -
A mode of execution in which special CID tables are generated during compilation and in which programs are executed under CID control. Debug mode is initiated by the DEBUG(ON) control statement, and terminated by the DEBUG(OFF) control statement.

Debug Session -
A sequence of interactions between you and CID, beginning when execution of your program is initiated in debug mode, and ending when a QUIT command is executed.

Group -
A CID command sequence established and assigned a name by a SET,GROUP command and executed when a READ command is issued.

Home Program -
The program unit in which variables, line numbers, and procedure names referenced in CID commands are assumed to be located unless appropriate qualifiers appear. By default, the home program is the program unit being executed when CID gains control. You can change the default with the SET,HOME command.

Interactive -
Job processing in which you and the system communicate with each other, rather than processing in which you submit a job and receive output later.

Interactive Mode -
The normal mode of CID execution. You enter commands directly from the terminal and CID immediately executes the commands. CID can also execute in batch mode.

Interrupt (noun) -
A control signal that you issue from the terminal. If your program is executing when CID detects an interrupt, an INTERRUPT trap occurs; if a CID command sequence is executing, the command sequence is suspended and you gain control.

On NOS, CID interprets both the user-break-1 and the user-break-2 terminal keys as the interrupt key. The user-break-1 and user-break-2 keys differ, depending on the terminal type (see the Network Products Interactive Facility reference manual). On most terminals, these keys are CONTROL P and CONTROL T, respectively; you can issue an interrupt by pressing CONTROL P (or CONTROL T) followed by a carriage return.

On NOS/BE, you can issue an interrupt by pressing %A followed by a carriage return (see the INTERCOM reference manual).

Interrupt (verb) -
To stop a running program in such a way that it can be resumed at a later time.

Module -
A named section of coding or data. An object module is output from a compiler or assembler. A source module is written by a programmer as input to a compiler or assembler. The word module alone usually refers to an object module. The components of system libraries are also modules.

Program Unit -
A COBOL program or subprogram.

Terminal Session -
The sequence of interactions between you and a terminal which begins when you log in, and terminates when you log out. Contrast with Debug Session.

Trap (noun) -
A mechanism that detects the occurrence of a specified condition, suspends execution of your program at that point, and transfers control to CID.

Trap (verb) -
To suspend program execution and transfer control to CID upon the detection of a specified condition.

BATCH MODE DEBUGGING C

CYBER Interactive Debug (CID) is primarily intended for interactive use, but can be used in batch mode. Possible reasons for using batch mode include a potentially 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 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 by using the system text editor, or you can punch the commands on cards, include them as part of the job deck, and copy file INPUT to DBUGIN. Commands are punched or written in the same format as in interactive mode; each card contains a single CID command (or a sequence of commands separated by semicolons).

As in interactive execution, 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 your program. The program executes until a breakpoint or trap 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.

All commands are executed except when execution is impossible.

In batch mode, all output from CID is written to a file named DBUGOUT. 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 copy 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.

All of the CID commands described in this guide are valid in batch mode. You can set breakpoints and traps, define command sequences, display and alter program values, 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 you would normally enter from the terminal.

A suggested technique for batch mode debugging is to use only breakpoints and traps with bodies. This way, the commands to be executed on suspension of execution appear in the input stream immediately after the SET,BREAKPOINT or SET,TRAP command that caused suspension. In addition, only one GO command is required.

An example of a program to be debugged in batch mode (under NOS) is illustrated in figure C-1. (To execute this program under NOS/BE, replace the job, user, and charge statements with a job statement containing the appropriate accounting information.) The contents of the output file DBUGOUT are shown in figure C-2.

In this batch example, the output options in the SET,OUTPUT command make the results in file DBUGOUT readable. The T option causes the SET,BREAKPOINT commands (and the commands that make up the breakpoint bodies) to be written to DBUGOUT when the breakpoints are established. The B option causes the commands within the bodies to be written when they are executed. The R option is meaningless in this example; this option would have caused group and sequence file commands to be written when they were executed.

```
JOB STATEMENT.
USER STATEMENT.
CHARGE STATEMENT.
COPYBR,INPUT,DBUGIN.
DEBUG(ON)
COBOL5.
LGO.
REWIND,DBUGOUT.
COPY,DBUGOUT,OUTPUT.
7/8/9 in column 1
SET,OUTPUT,E,W,I,D,T,B,R
SET,BREAKPOINT,PR.SORTING [
DISPLAY BID OF BID-TABLE (1)
DISPLAY BID OF BID-TABLE (2)
DISPLAY BID OF BID-TABLE (3)
DISPLAY BID OF BID-TABLE (4)
]
SET,BREAKPOINT,L.66 [
DISPLAY SORT-RECORD
]
SET,BREAKPOINT,PR.SORT-OUT-PROC [
DISPLAY SORT-RECORD
]
GO
QUIT
7/8/9 in column 1
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  SORT-BIDS-2.
      *
      *    THIS PROGRAM SORTS A LIST OF BIDS SUBMITTED FOR ONE ITEM
      *    AT AN AUCTION.  EACH INPUT LINE TAKES THE FORM:
      *      BID                       PICTURE 9999V99.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.  CYBER-170.
       OBJECT-COMPUTER.  CYBER-170.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT IN-FILE ASSIGN TO "INPUT".
           SELECT OUT-FILE ASSIGN TO "OUTPUT".
           SELECT SORT-FILE ASSIGN TO SFILE.
       DATA DIVISION.
       FILE SECTION.
       FD IN-FILE
           LABEL RECORD IS OMITTED
           DATA RECORD IS LINE-IN.
       01  LINE-IN.
           05  BID                     PICTURE 9999V99.
           05  FILLER                  PICTURE X(4).
       FD OUT-FILE
           LABEL RECORD IS OMITTED
           DATA RECORD IS LINE-OUT.
       01  LINE-OUT.
           05  FILLER                  PICTURE X(10).
           05  BID                     PICTURE $9999.99.
       SD  SORT-FILE
           RECORD CONTAINS 6 CHARACTERS
           DATA RECORD IS SORT-RECORD.
       01  SORT-RECORD.
           05  BID                     PICTURE 9999V99.
```

Figure C-1. Card Deck for Batch Debug Session (Sheet 1 of 2)

```
               WORKING-STORAGE SECTION.
               01  BID-INFORMATION.
                   05  NUMBER-OF-BIDS          PICTURE 99V.
                   05  BID-TABLE               OCCURS 10 TIMES
                                               INDEXED BY BID-INDEX.
                       10  BID                 PICTURE 9999V99.
               PROCEDURE DIVISION.
               INITIALIZATION SECTION.

               OPEN-FILES.
                   OPEN INPUT IN-FILE
                        OUTPUT OUT-FILE.
               INITIALIZE-VALUES.
                   MOVE ZERO TO NUMBER-OF-BIDS.
               PROCESS-A-BID SECTION.
               READ-BIDS.
                   READ IN-FILE AT END GO TO SORTING.
                   ADD 1 TO NUMBER-OF-BIDS.
                   MOVE  LINE-IN TO BID OF BID-TABLE (NUMBER-OF-BIDS).
                   GO TO READ-BIDS.
               SORTING SECTION.
               SORT-THE-BIDS.
                   SORT SORT-FILE
                           ON DESCENDING KEY BID OF SORT-RECORD
                         INPUT PROCEDURE IS SORT-IN-PROC
                         OUTPUT PROCEDURE IS SORT-OUT-PROC.
                   GO TO WRITE-RESULTS.
               SORT-IN-PROC SECTION.
               START-OF-SECTION.
                   PERFORM VARYING BID-INDEX FROM 1 BY 1
                       UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS
                   RELEASE SORT-RECORD FROM BID OF BID-TABLE (BID-INDEX)
                   END-PERFORM.
               SORT-OUT-PROC SECTION.
               START-OF-SECTION.
                   PERFORM SORTING-PARAGRAPH VARYING BID-INDEX FROM 1 BY 1
                       UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
                   GO TO END-OF-SECTION.
               SORTING-PARAGRAPH.
                     RETURN SORT-FILE RECORD
                         AT END GO TO END-OF-SECTION
                     MOVE SORT-RECORD TO BID OF BID-TABLE (BID-INDEX).
               END-OF-SECTION.
               WRITE-RESULTS SECTION.
               WRITE-BIDS.
                   PERFORM WRITE-ONE-BID VARYING BID-INDEX FROM 1 BY 1
                       UNTIL BID-INDEX IS GREATER THAN NUMBER-OF-BIDS.
                   GO TO END-OF-RUN
               WRITE-ONE-BID.
                   MOVE BID OF BID-TABLE (BID-INDEX) TO BID OF LINE-OUT.
                   WRITE LINE-OUT.
               END-OF-RUN SECTION.
               CLOSE-FILES.
                   CLOSE IN-FILE, OUT-FILE.
               STOP-RUN.
                   STOP RUN.
        7/8/9 in column 1
        444332
        011023
        648234
        003325
        6/7/8/9 in column 1
```

Figure C-1. Card Deck for Batch Debug Session (Sheet 2 of 2)

```
                                                CYBER INTERACTIVE DEBUG 1.2-552.


          CYBER INTERACTIVE DEBUG
          SET,OUTPUT,E,W,I,D,T,B,R
          SET,BREAKPOINT,PR.SORTING [
          IN COLLECT MODE
          DISPLAY BID OF BID-TABLE (1)
          DISPLAY BID OF BID-TABLE (2)
          DISPLAY BID OF BID-TABLE (3)
          DISPLAY BID OF BID-TABLE (4)
          ]
          END COLLECT
          SET,BREAKPOINT,L.66 [
          IN COLLECT MODE
          DISPLAY SORT-RECORD
          ]
          END COLLECT
          SET,BREAKPOINT,PR.SORT-OUT-PROC [
          IN COLLECT MODE
          DISPLAY SORT-RECORD
          ]
          END COLLECT
          GO
          DISPLAY BID OF BID-TABLE (1)
           4443.32
          DISPLAY BID OF BID-TABLE (2)
           110.23
          DISPLAY BID OF BID-TABLE (3)
           6482.34
          DISPLAY BID OF BID-TABLE (4)
           33.25
          ]
          DISPLAY SORT-RECORD

          ]
          DISPLAY SORT-RECORD
          444332
          ]
          DISPLAY SORT-RECORD
          011023
          ]
          DISPLAY SORT-RECORD
          648234
          ]
          DISPLAY SORT-RECORD
          003325
          ]
          *T #17, END IN L.90
          QUIT
```

Figure C-2. Listing of File DBUGOUT

SUMMARY OF CID COMMANDS D

This section provides a summary of the CID commands described in this user's guide. (This user's guide describes a subset of the CID commands available; see the CID reference manual for a description of all available CID commands.)

CID commands are divided into two types: language-dependent and language-independent commands. The language-dependent commands are similar in format and action to statements in the language in which the home program is written; language-independent commands are the same in format and action for all programming languages.

LANGUAGE-DEPENDENT COMMANDS

Language-dependent commands are similar in format and action to statements in the language in which the home program was written. All of the language-dependent commands described in this manual are COBOL CID commands. An example of a language-dependent command is the MOVE command described in section 3. The MOVE command has the following form:

```
MOVE value TO identifier-1
```

This command is a restricted form of the COBOL MOVE statement; therefore, it is a language-dependent CID command. Table D-1 shows the language-dependent commands described in this guide, and the pages on which they are described.

TABLE D-1. LANGUAGE-DEPENDENT COMMANDS

Command Name	Page	Description
DISPLAY	3-14	Displays the values of data items and literals
GO TO	3-8	Resumes execution at a paragraph or section
MOVE	3-15	Changes the value of a data item
SET	3-15	Changes the value of an index

Each of the COBOL CID commands has the same name as a language-independent command. If you specify a comma after a COBOL CID command name, CID assumes the command is a language-independent command, and the results can be unpredictable. See the subsection Language-Independent Commands for a further discussion of this problem.

LANGUAGE-INDEPENDENT COMMANDS

Language-independent commands have a format designed specifically for use with CID. These commands appear as follows:

```
command-name,parameter-list
```

When the home program is a COBOL program, a comma must follow the command name. Parameters in the parameter list can be separated by commas or spaces.

Many language-independent commands have a short form that can be used in place of the command name. When the short form is used, the comma is optional; a space can separate the short form from the parameter list.

Table D-2 shows the language-independent commands described in this guide and the pages on which they are described.

Many of the language-independent commands have the same names as COBOL CID commands. When you enter the name of one of these commands, CID must determine whether you have entered the COBOL CID command or the language-independent command.

Assuming the home program is a COBOL program, you can force CID to interpret the command as a language-independent command by entering a comma after the command name or by entering the short form of the command. Otherwise, CID interprets the command as a COBOL CID command.

TABLE D-2. LANGUAGE-INDEPENDENT COMMANDS

Command Name	Short Form	Page	Description
CLEAR,AUXILIARY	CAUX	3-20	Closes the auxiliary output file
CLEAR,BREAKPOINT	CB	3-3	Removes breakpoints
CLEAR,GROUP	CG	5-5	Removes groups
CLEAR,OUTPUT	COUT	3-20	Turns off CID output to the terminal
CLEAR,TRAP	CT	3-7	Removes traps
DISPLAY†	D	3-17	Displays debug variables
EXECUTE	EXEC	5-8	Resumes program execution
GO		2-3	Resumes execution of your program or of a suspended command sequence
HELP		2-4	Displays information about CID commands
LIST,BREAKPOINT	LB	3-3	Displays defined breakpoints
LIST,GROUP	LG	5-5	Displays defined groups
LIST,MAP	LM	4-7	Displays load map information
LIST,STATUS	LS	3-18	Displays information about the status of the debug session
LIST,TRAP	LT	3-7	Displays defined traps
LIST,VALUES	LV	3-9	Displays program values
PAUSE		5-8	Suspends execution of a command sequence
QUIT		2-3	Terminates the debug session
READ		5-4	Executes a group or a sequence of commands stored on a file; defines groups and traps previously saved
SAVE,BREAKPOINT	SAVEB	5-10	Saves breakpoint definitions onto a file
SAVE,GROUP	SAVEG	5-10	Saves group definitions onto a file
SAVE,TRAP	SAVET	5-10	Saves trap definitions onto a file
SAVE,*		5-10	Saves all breakpoint, group, and trap definitions onto a file
SET,AUXILIARY	SAUX	3-20	Establishes an auxiliary output file
SET,BREAKPOINT	SB	3-2	Establishes a breakpoint
SET,GROUP	SG	5-4	Establishes a group
SET,HOME	SH	4-1	Designates the home program
SET,OUTPUT	SOUT	3-19	Selects output types to be displayed at the terminal
SET,TRAP	ST	3-6	Establishes a trap
STEP	S	3-8	Executes a few statements or procedures
SUSPEND		5-11	Suspends the debug session

†The language-independent DISPLAY command is called the D command in this guide.

INDEX

COMMENT SHEET

MANUAL TITLE: CYBER Interactive Debug Version 1
Guide for Users of COBOL Version 5

PUBLICATION NO.: 60484120

REVISION: B

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments on the back (please include page number references).

_____ Please reply _____ No reply necessary

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

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.
FOLD ON DOTTED LINES AND TAPE

NAME:

COMPANY:

STREET ADDRESS:

CITY/STATE/ZIP:

TAPE TAPE

102680323