

CA-IDMS[®]

Online Debugger
15.0



Computer Associates™

This documentation and related computer software program (hereinafter referred to as the "Documentation") is for the end user's informational purposes only and is subject to change or withdrawal by Computer Associates International, Inc. ("CA") at any time.

THIS DOCUMENTATION MAY NOT BE COPIED, TRANSFERRED, REPRODUCED, DISCLOSED, OR DUPLICATED, IN WHOLE OR IN PART, WITHOUT THE PRIOR WRITTEN CONSENT OF CA. THIS DOCUMENTATION IS PROPRIETARY INFORMATION OF CA AND PROTECTED BY THE COPYRIGHT LAWS OF THE UNITED STATES AND INTERNATIONAL TREATIES.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO THE END USER OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED OF SUCH LOSS OR DAMAGE.

THE USE OF ANY PRODUCT REFERENCED IN THIS DOCUMENTATION AND THIS DOCUMENTATION IS GOVERNED BY THE END USER'S APPLICABLE LICENSE AGREEMENT.

The manufacturer of this documentation is Computer Associates International, Inc.

Provided with "Restricted Rights" as set forth in 48 C.F.R. Section 12.212, 48 C.F.R. Sections 52.227-19(c)(1) and (2) or DFARS Section 252.227.7013(c)(1)(ii) or applicable successor provisions.

First Edition, December 2000

© 2000 Computer Associates International, Inc.
One Computer Associates Plaza, Islandia, NY 11749
All rights reserved.

All trademarks, trade names, service marks, or logos referenced herein belong to their respective companies.

Contents

How to use this manual	v
Chapter 1. Introduction	1-1
1.1 About the debugger	1-3
1.2 Debugger features	1-4
1.3 Debugging process	1-6
1.3.1 Prompt mode	1-6
1.3.2 Menu mode	1-8
1.3.3 Setup phase	1-11
1.3.4 Runtime phase	1-11
1.4 Session considerations	1-13
1.4.1 Performance standards	1-13
1.4.2 Valid breakpoints	1-14
1.4.3 Program currency	1-14
Chapter 2. Command Considerations	2-1
2.1 About this chapter	2-3
2.2 Expression components	2-4
2.2.1 Debugger symbols	2-4
2.2.2 User symbols	2-6
2.2.3 Program symbols	2-7
2.2.3.1 Data field names	2-7
2.2.3.2 Line numbers	2-8
2.2.3.3 Qualifying program symbols	2-8
2.2.4 Expression operators	2-9
2.3 Length attributes	2-11
2.3.1 Expressions with data characteristics	2-11
2.3.2 Expressions without data characteristics	2-12
2.4 Parsing rules	2-14
2.5 Command modification	2-15
2.6 Delimiters	2-16
2.7 Data values	2-17
2.8 Command format	2-18
Chapter 3. Debugger Commands	3-1
3.1 Summary of commands	3-3
3.2 AT	3-4
3.3 DEBUG	3-7
3.4 EXIT	3-9
3.5 IOUSER	3-10
3.6 LIST	3-11
3.7 MENU	3-14
3.8 PROMPT	3-15
3.9 QUALIFY	3-16
3.10 QUIT	3-18
3.11 RESUME	3-19
3.12 SET	3-20

3.13	SNAP	3-24
3.14	WHERE	3-26
Chapter 4. Debugging in Menu Mode		4-1
4.1	Features of menu mode	4-3
4.2	Screen design	4-4
4.2.1	Heading area	4-4
4.2.2	Display area	4-7
4.2.3	Specification area	4-7
4.2.4	Selection area	4-7
4.3	Accessing screens	4-9
4.3.1	Screen hierarchy	4-9
4.3.2	Screen sequence	4-10
4.3.3	Selection processing	4-10
4.3.4	Command currency	4-11
4.4	Activity screens	4-12
4.4.1	At screen	4-12
4.4.2	Debug screen	4-13
4.4.3	List screen	4-14
4.4.4	Resume screen	4-15
4.4.5	Set screen	4-16
4.4.6	Snap screen	4-17
4.5	Global help screens	4-19
4.5.1	Usage screen	4-19
4.5.2	Symbols screen	4-19
4.5.3	Keys screen	4-20
Chapter 5. Aids for Debugging Assembler, COBOL, and PL/I Programs		5-1
5.1	Overview	5-3
5.2	Compiler options	5-4
5.3	COBOL programs	5-5
5.3.1	Preliminary computations	5-5
5.3.2	Sample COBOL online debugger session	5-9
5.4	PL/I programs	5-11
5.4.1	Preliminary computations	5-11
5.4.2	Sample PL/I online debugger session	5-14
Index		X-1

How to use this manual

Contents

This manual provides detailed instructions for users debugging programs that operate in a CA-IDMS/DC and CA-IDMS/UCF environment. The manual contains four chapters:

- **Introduction** — General presentation of the debugger
- **Command Considerations** — Description of the expressions that can be used in debugger commands, the length attributes of debug expressions, and user-supplied data values
- **Debugger Commands** — Detailed discussion of each of the debugger commands
- **Debugging in Menu Mode** — Detailed description of screens and how to use them for debugging

For more information

For further information related to this manual, see:

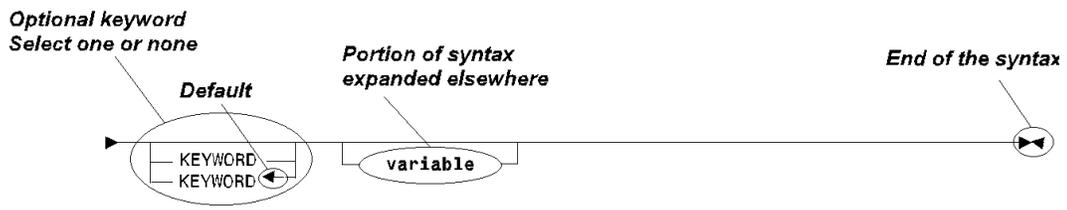
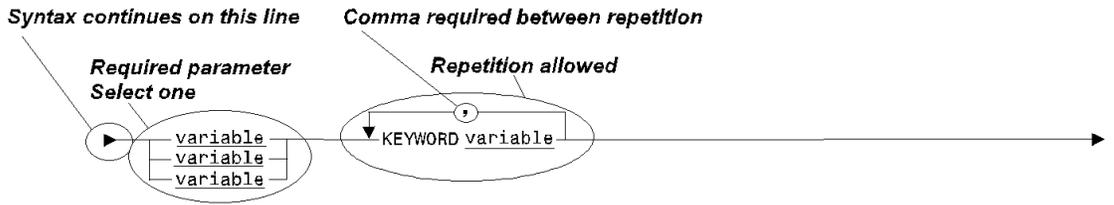
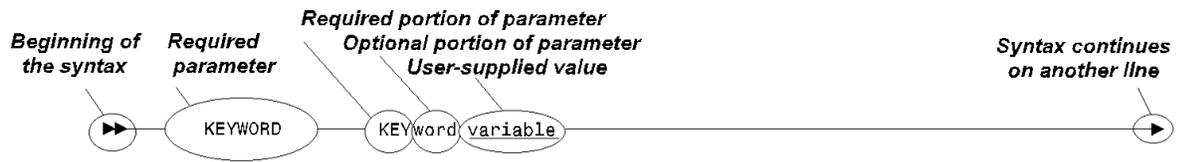
- *CA-IDMS System Generation*
- *CA-IDMS System Operations*
- *CA-ADS Reference Guide*
- *CA-IDMS Navigational DML Programming*
- *CA-IDMS Security Administration*

Understanding syntax diagrams

Look at the list of notation conventions below to see how syntax is presented in this manual. The example following the list shows how the conventions are used.

UPPERCASE OR SPECIAL CHARACTERS	Represents a required keyword, partial keyword, character, or symbol that must be entered completely as shown.
lowercase	Represents an optional keyword or partial keyword that, if used, must be entered completely as shown.
<u>underlined lowercase</u>	Represents a value that you supply.
←	Points to the default in a list of choices.
lowercase bold	Represents a portion of the syntax shown in greater detail at the end of the syntax or elsewhere in the document.
▶▶	Shows the beginning of a complete piece of syntax.
◀◀	Shows the end of a complete piece of syntax.
▶	Shows that the syntax continues on the next line.
▶	Shows that the syntax continues on this line.
▶	Shows that the parameter continues on the next line.
▶	Shows that a parameter continues on this line.
▶ parameter ▶	Shows a required parameter.
▶ parameter parameter ▶	Shows a choice of required parameters. You must select one.
▶ parameter ▶	Shows an optional parameter.
▶ parameter parameter ▶	Shows a choice of optional parameters. Select one or none.
▶ parameter ▶	Shows that you can repeat the parameter or specify more than one parameter.
▶ parameter , parameter ▶	Shows that you must enter a comma between repetitions of the parameter.

Sample syntax diagram



Chapter 1. Introduction

1.1 About the debugger	1-3
1.2 Debugger features	1-4
1.3 Debugging process	1-6
1.3.1 Prompt mode	1-6
1.3.2 Menu mode	1-8
1.3.3 Setup phase	1-11
1.3.4 Runtime phase	1-11
1.4 Session considerations	1-13
1.4.1 Performance standards	1-13
1.4.2 Valid breakpoints	1-14
1.4.3 Program currency	1-14

1.1 About the debugger

What you can debug: The CA-IDMS online debugger is an interactive facility used to detect, trace, and resolve programming errors in programs that run under the control of CA-IDMS/DC and CA-IDMS/UCF. The debugger can be used with these load modules:

- Assembler, COBOL, and PL/I programs
- CA-ADS
- Subschemas
- Maps
- Tables

►► For more information on using the debugger with Assembler, COBOL, and PL/I programs, see Chapter 5, “Aids for Debugging Assembler, COBOL, and PL/I Programs” on page 5-1.

How you use the debugger: You use the online debugger to:

- Receive control when an abend occurs

The online debugger receives control when your program abends (for example, with a data exception). You can then determine the abending instruction and examine program variable storage to determine the error.

- Receive control at predetermined breakpoints

In order to trap logic errors, you set breakpoints that halt program execution at a specified line number. The online debugger receives control when your program reaches that line number, so that you can examine program variable storage.

Chapter contents: This introductory chapter discusses:

- Debugger features
- Debugging in prompt or menu mode
- Setup and runtime phases of a debugger session
- Factors to consider when establishing a debugger session

1.2 Debugger features

High level of control: The online debugger allows you to maintain a high level of control over the debugging process. With the debugger, you can:

- Set breakpoints
- Display the contents of registers and storage
- Modify storage values
- Snap tasks and storage areas to the log
- Trap abends in the module being debugged

Each of these functions is discussed below.

Setting breakpoints: **Breakpoints** are temporary program interruptions that you can set at any address within a program or dialog that complies with debugger validation rules, as described in "Valid Breakpoints" later in this chapter.

At runtime, the debugger takes control at these breakpoints, and program execution is temporarily suspended. While execution is suspended, you can perform a variety of activities before returning control to the DC/UCF system or resuming execution of the program.

Displaying and modifying storage values: You can examine storage values in any area, assuming that you have the security necessary to access the area. (Traditional error-handling routines and dumps supply information only after an error occurs or a program finishes executing.)

You can modify storage values and then execute the program to test the modifications.

The ability to examine and modify storage values in any area makes the debugger a very powerful tool.

Therefore, it's important to use debugger security to control access to storage.

►► For information on the security methods used by the debugger, see *CA-IDMS Security Administration*.

Snapping tasks and storage areas: You can create dumps for a task or for a specific area; the dumps are written to the DC/UCF log. From the log you can make a hard copy of storage contents and then examine them at your leisure.

Trapping abends: The debugger automatically takes control when an instruction causes an abend in the module being debugged, allowing you to examine storage and to take appropriate action.

Managing program execution: The debugger also provides you with a flexible tool for managing an executing program. Under the control of the debugger during runtime:

- After a breakpoint, you can:
 - Allow the program to resume execution from the current breakpoint address
 - Specify resumption at an address before or after the breakpoint
- After an abend, you can:
 - Allow standard abend processing to continue
 - Resume program execution at an address before or after the abend
- In both cases, you can modify previous debugger commands or issue new commands, for example to:
 - Ignore all remaining breakpoints
 - Bypass specific breakpoints
 - Set additional breakpoints for the duration of a session

1.3 Debugging process

What to define: You cannot debug an Assembler, COBOL, or PL/I program until you define it to the DC/UCF system. For example, you cannot debug a program until it is defined in the PROGRAM statement at system generation time or defined dynamically with the DCMT VARY DYNAMIC PROGRAM statement.

Similarly, you must define the program task code either in the TASK statement at system generation or dynamically with the DCMT VARY DYNAMIC TASK statement.

Important: You don't have to define the task code for the initial stage of the debugging process, but you must define it before executing the program.

You don't have to define CA-ADS dialogs, subschemas, maps, and tables.

Debugger structure: You can conduct a debugger session in one of two modes or a combination of both:

- **Prompt mode** enables you to issue debugger commands line by line
- **Menu mode** enables you to issue commands from a series of activity and tutorial screens

Debugging a module takes place in two phases:

- The **setup phase**, invoked before a program is executed
- The **runtime phase**, occurring during program execution and dependent on actions taken during setup

DEBUG and QUIT: A debugger session begins when you issue the first DEBUG task code. A session ends when you either issue the debugger QUIT command or terminate the DC/UCF session by signing off.

Debugger session modes and phases are discussed in detail below.

1.3.1 Prompt mode

Line-oriented method: Prompt mode is the line-oriented method of communicating with the debugger. In prompt mode you can:

- Initiate a debugging session
- Issue a debugger command
- Return to the DC/UCF system

Initiating a debugging session: To initiate a debugging session in prompt mode, enter the DEBUG task code in response to the Enter Next Task Code prompt:

```
ENTER NEXT TASK CODE:  
debug
```

The debugger indicates that it is in control by responding with its special prompt:

```
DEBUG >
```

Issuing a debugger command: You can issue debugger commands whenever the debugger responds with the DEBUG> prompt. **To issue a debugger command at the same time you initiate a debugging session**, enter the task code in conjunction with the DEBUG command that names the entity to be debugged. In this example, the task code DEBUG is followed by a DEBUG command that identifies TESTPROG to the debugger:

```
ENTER NEXT TASK CODE:  
debug debug testprog
```

When you enter the above command, you invoke the debugging facility. The command is echoed, and the debugger responds by validating the command and displaying the next DEBUG> prompt:

```
DEBUG TESTPROG  
DEBUG > DEBUGGING INITIATED FOR TESTPROG VERSION 1  
DEBUG >
```

If you try to debug a program which has not been defined to a DC/UCF system, the debugger issues an error message after echoing the command, then repeats the command that cannot be completed, and redisplay the DEBUG> prompt, as in this example:

```
DEBUG TESTPROG  
DC574902 DEBUG > LOAD OF TESTPROG FAILED - NOT FOUND  
DEBUG > DEBUG TESTPROG  
DEBUG >
```

Difference between EXIT and QUIT: To return control to the DC/UCF system, issue either the EXIT command or the QUIT command.

The **EXIT** command saves the debugger control blocks and allows you to continue the same debugger session.

The **QUIT** command clears the control blocks and terminates the debugger session completely.

How to check session activity: To determine whether a debugger session exists, issue the command DCMT DISPLAY LTE *. This command lists information about your logical terminal. If you see DEBUG ACT, a debugger session is active; if you see DEBUG INACT, no debugger session is active.

To inquire for a list of modules known to the debugger, use the DEBUG INQUIRE command (see Chapter 3, “Debugger Commands” on page 3-1).

Valid commands: In prompt mode, you can use all commands except RESUME, IOUSER, and WHERE during setup, and all commands except DEBUG during runtime. The PROMPT command performs no function while you are in prompt mode.

►► For a detailed discussion of the debugger commands, see Chapter 3, “Debugger Commands” on page 3-1.

1.3.2 Menu mode

Choosing activities from screens: Menu mode is designed to make your options easy to see. You can enter commands or display information by filling in the fields on a series of fixed-format screens:

- Activity screens provide fields for commands that require additional input
- Individual help screens provide detailed descriptions of each command
- The Usage global help screen summarizes debugging activities
- Two other global help screens let you display program and debugger symbols and program function key (PF-key) assignments

►► For a complete description of each of the screens, see Chapter 4, “Debugging in Menu Mode” on page 4-1.

Initiating a debugger session: To initiate a debugging session in menu mode, issue the DEBUG task code followed by the MENU command in response to the Enter Next Task Code prompt:

```
ENTER NEXT TASK CODE:
debug menu
```

When you enter this command, you see the Usage screen, which is the top-level menu screen:

```

      IDMS-DC REL 15.0 ONLINE DEBUGGER *** USAGE ***          SETUP  PAGE 1 OF 4
PROGRAM:          V:          CSECT:
->
      PROCEDURAL COMMANDS.

EXIT.....RETURNS CONTROL TO IDMS-DC/UCF WITHOUT TERMINATING THE CURRENT DEBUGGER
SESSION.
QUIT.....TERMINATES THE DEBUGGER SESSION AND RETURNS CONTROL TO IDMS-DC/UCF.
PROMPT...INVOKES THE PROMPT MODE OF THE DEBUGGER.

      RETRIEVAL COMMANDS.

AT.....ESTABLISHES OR MODIFIES BREAKPOINTS WITHIN A USER PROGRAM.
DEBUG....DESIGNATES, DURING THE SETUP PHASE, THE ENTITY TO BE DEBUGGED OR
INQUIRES ABOUT ENTITIES KNOWN TO THE DEBUGGER.
IOUSER...DISPLAYS THE USER SCREEN THAT IS CURRENT WHEN A BREAKPOINT, PROGRAM
INTERRUPT OR TRAPPED ABEND IS ENCOUNTERED.

NEXT _ ACTIVITY OR _ HELP:
  _ AT      _ LIST  _ SET   _ SNAP  _ RESUME  _ DEBUG  _ WHERE

      EXIT      _ PROMPT  _ QUIT   _ IOUSER
HELP SCREENS:  _ USAGE  _ SYMBOLS _ KEYS
```

Switching mode: To switch from prompt mode to menu mode, issue the MENU command in response to the DEBUG> prompt:

```
DEBUG >
menu
```

Now you also see the Usage screen.

Going to a specific screen: To go to a specific activity screen or global help screen, issue the MENU command followed by a valid screen name. This example illustrates the use of the DEBUG task code with a MENU command that names the screen to be displayed:

```
ENTER NEXT TASK CODE:
debug menu at
```

When you enter the above command, you invoke the debugging facility in menu mode and see the At command activity screen:

```

      IDMS-DC REL 15.0 ONLINE DEBUGGER *** AT ***          SETUP   PAGE 1 OF 1
PROGRAM:          V:          CSECT:
->

ADD BREAKPOINT AT:
BEFORE: MAX          AFTER: 0          EVERY: 1

OTHER ACTION.....: (I-INQUIRE/D-DELETE/G-IGNORE)
BREAKPOINT OR <ALL>:

NEXT _ ACTIVITY OR _ HELP:
  _ AT      _ LIST  _ SET    _ SNAP  _ RESUME  _ DEBUG  _ WHERE
  _ EXIT    _ PROMPT _ QUIT   _ IOUSER
HELP SCREENS: _ USAGE _ SYMBOLS _ KEYS
```

Valid commands: Menu mode allows the same set of debugger commands as prompt mode, with the exception that the PROMPT command is allowed and the MENU command is disabled.

Leaving menu mode: To leave menu mode you can:

- Select the PROMPT activity
- Return to prompt mode with the associated control key
- Enter the PROMPT command on the menu DEBUG> prompt line

Note: If you leave menu mode with an EXIT command or with the CLEAR control key, the debugger remains in menu mode. Subsequently, when control returns

to the debugger, the debugger is still in menu mode. The debugger remains in menu mode until you issue the PROMPT command.

►► For a more detailed discussion of menu mode, see Chapter 4, “Debugging in Menu Mode” on page 4-1.

1.3.3 Setup phase

Breakpoints and abends: The setup phase is the preliminary phase of the debugging process. During this stage, you can define modules to the debugger for two reasons:

- **To enable the setting of breakpoints**

Breakpoints can be established as soon as the `DEBUG` command is used to define the load module to the debugger.

- **To gain control under the debugger when a program check or abend occurs**

The setting of breakpoints is not mandatory; you can trap possible abends in a program during runtime and receive control under the debugger if:

- You have defined the program to the debugger (that is, issued a `DEBUG` command for the program during the setup phase)
- You have defined the current DC/UCF program to the debugger

The last program to receive control through a `#LINK` or `#XCTL` is called the **current DC/UCF program**. When a program check occurs in a module unknown to the debugger, you will gain control under the debugger if the current DC/UCF program is defined to the debugger.

►► For a detailed discussion of DC/UCF and debugger methods of assigning currency, see 1.4.3, “Program currency” on page 1-14.

1.3.4 Runtime phase

DEBUG and EXIT required: The runtime phase of the debugging process takes place during the execution of a program. Debugging cannot occur during runtime unless:

- You have used the **DEBUG command** during the setup phase to define the program to the debugger
- You have used the **EXIT command**, which retains the debugger control blocks, when leaving the setup phase

What happens at the breakpoint: When you have defined a program to the debugger, the program task code invokes both the runtime phase of the debugger and the execution of the program. At a breakpoint, the DC/UCF runtime system suspends program execution, and you gain control under the debugger. A message is displayed that signals the breakpoint interrupt and describes its location.

Three breakpoint display formats: For example, assume that a program called `TESTPROG` is defined to the debugger and a breakpoint is established like this during the setup phase:

```
DEBUG >  
at @00bf080
```

The debugger verifies the establishment of the breakpoint:

```
AT @00BF080
AT > @00BF080 ADDED
DEBUG >
```

When this breakpoint is encountered during runtime, the debugger identifies the address, the program, and the debug expression that established the breakpoint:

```
AT OFFSET 080 IN TESTPROG EXPRESSION @00BF080
DEBUG >
```

In response to the DEBUG> prompt, you can make additional queries or perform other debugging activities.

1.4 Session considerations

Three factors: You'll need to consider the following factors when you establish and conduct debugger sessions:

- Performance standards
- Valid breakpoints
- Program currency

Each of these topics is discussed below.

1.4.1 Performance standards

All activities permissible: During a debugger session, you can perform any activity related to DC/UCF, not just debugging. For a given session, there are no restrictions on the number or kinds of entities debugged or on the length of the session.

For example, within a single debugger session, you can successively:

- Initiate a debugger setup phase
- Leave the debugger setup phase to conduct an online PLOG session
- Return to the setup phase to debug another program
- Leave the debugger setup phase again to conduct an IDD session
- Execute one of the programs you are debugging

Minimize unrelated work: When the DEBUG task code initiates a debugger session, the DC/UCF system saves your current screen, whether or not the screen is directly related to any modules being debugged. Consequently, the debugger incurs some processing overhead each time the current screen changes. For best performance, therefore, keep work unrelated to the debugging process to a minimum.

Also, although the setup phase is pseudoconversational, the runtime phase is completely conversational, which ties up system resources. Even database resources are tied up while the debugger has control.

In order to use resources most efficiently, therefore, always return control to DC/UCF before you leave your terminal or attend to concerns other than debugging.

1.4.2 Valid breakpoints

Verified by debugger: Program breakpoints, established with the AT command, must be set at addresses that contain valid instructions or valid command elements (CMEs for CA-ADS dialogs). If the address cannot be validated, the debugger displays a message to indicate that the breakpoint could not be set. A verifying message is displayed when the address is valid.

CAUTION: The debugger checks for a valid operation code at each breakpoint that is set; you are responsible for placing the breakpoint at an actual instruction. If a breakpoint address resolves to an address offset that contains a valid operation code but does not contain a valid instruction, the program could be altered with unpredictable results.

1.4.3 Program currency

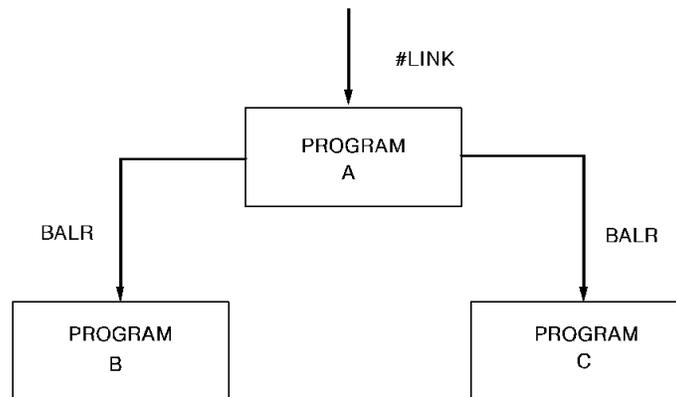
Determines abend trapping: When a task abends or when a program check occurs, the setting of program currency determines whether or not the debugger traps the abend and transfers control to you.

DC/UCF and debugger currency: The DC/UCF system assigns currency on the basis of the most recent program to have been given control with #LINK or #XCTL program control services.

The **debugger** assigns currency according to these rules:

- If the address of the interrupt is contained in one of the programs defined to the debugger, this program is assigned debugger currency, and you are given control under the debugger
- If the address is not found in a debugged program, the debugger checks the current IDMS-DC/UCF program to see whether it has been defined to the debugger:
 - If the current DC/UCF program has been defined to the debugger, this program is assigned debugger currency, and you gain control under the debugger
 - If the program has not been defined, no debugger currency is assigned, you do not gain control under the debugger, and the standard DC/UCF abend processing takes place

Sample program structure: The following examples illustrate how program currency can affect whether the DC/UCF system passes control to the debugger. Each of the examples is based on the sample program structure:

Sample program structure for examples

Example 1: During the setup phase, you define Programs A, B, and C to the debugger. When the program is executing, a program check occurs in Program B.

These currencies are now in effect:

- The current DC/UCF program is Program A, the last program to have been given control by #LINK or #XCTL
- Debugger currency is assigned to Program B

You receive control under the debugger because Program B, one of the programs defined to the debugger, contains the address of the interrupt.

Example 2: During the setup phase, you define Program A to the debugger. When the program is executing, a program check occurs in Program B.

These currencies are now in effect:

- The current DC/UCF program is Program A, the last program to have been given control by #LINK or #XCTL
- Debugger currency is assigned to Program A

You receive control under the debugger because the current DC/UCF program has also been defined to the debugger.

Example 3: During the setup phase, you define Program C to the debugger. When the program is executing, a program check occurs in Program B.

These currencies are now in effect:

- The current DC/UCF program is Program A, the last program to have been given control by #LINK or #XCTL
- Debugger currency is not assigned, because the debugger cannot find the interrupt address in a known program and the current DC/UCF program is not defined to the debugger

You do not receive control under the debugger because no debugger currency can be set. The program abends without an interruption from the debugger, and the system issues a standard abend message.

Example 4: During the setup phase, you define Program A to the debugger. During execution, Program B branches into unknown storage and a program check occurs.

These currencies are now in effect:

- The current DC/UCF program is Program A, the last program to have been given control by #LINK or #XCTL
- Debugger currency is assigned to Program A

You receive control under the debugger because the current DC/UCF program has also been defined to the debugger.

Chapter 2. Command Considerations

- 2.1 About this chapter 2-3
- 2.2 Expression components 2-4
 - 2.2.1 Debugger symbols 2-4
 - 2.2.2 User symbols 2-6
 - 2.2.3 Program symbols 2-7
 - 2.2.3.1 Data field names 2-7
 - 2.2.3.2 Line numbers 2-8
 - 2.2.3.3 Qualifying program symbols 2-8
 - 2.2.4 Expression operators 2-9
- 2.3 Length attributes 2-11
 - 2.3.1 Expressions with data characteristics 2-11
 - 2.3.2 Expressions without data characteristics 2-12
- 2.4 Parsing rules 2-14
- 2.5 Command modification 2-15
- 2.6 Delimiters 2-16
- 2.7 Data values 2-17
- 2.8 Command format 2-18

2.1 About this chapter

When issuing debugger commands, you consider:

Expression components	Variables that can be specified in a debug expression
Length attributes	Display lengths for expressions with and without data characteristics
Parsing rules	Debugger rules for processing command input
Command modification	Rules for modifying commands
Delimiters	Delimiters recognized by the debugger
Data values	Numeric and string values recognized by the debugger
Command format	Guidelines used to format a debugger command

This chapter discusses each of these topics.

2.2 Expression components

Four basic components: The basic components of a debug expression are:

- Debugger symbols
- User symbols
- Program symbols
- Operators

Three ways to appear: When a debug expression is used in a command, the expression can appear as:

- A single debugger symbol, user symbol, program symbol, or integer
- Multiple debugger symbols, user symbols, program symbols, and integers joined by operators
- Multiple expressions joined by operators

2.2.1 Debugger symbols

Three categories: Debugger symbols can:

- Designate general registers
- Designate certain DC/UCF system entities
- Point to specific addresses

General registers symbols: **General registers** include the registers used by the program at the time of execution and the registers used by the DC/UCF system. The program status word (PSW) and register definitions are always preceded by a colon (:) and are specified by these symbols:

- **:PSW** for the current program status word
- **:R n** for the user program register at the time of interrupt, where n represents the number of the register and can have a value of 0 through 15
- **:REGS** for all user program registers at the time of interrupt
- **:SR n** for a DC/UCF system register at the time of interrupt, where n represents the number of the register and can have a value of 0 through 15
- **:SREGS** for all DC/UCF system registers at the time of interrupt

Important: A single debug expression can reference only one general register.

DC/UCF system symbols: Certain DC/UCF system symbols also function as debugger entities, and you can refer to them during a debugging session. A colon (:) must precede each symbol. These are the valid symbols:

:BAT	Base address table for session
:CSA	DC/UCF common storage area
:DLB	Debug local block, control block required for debugging session
:LTE	Current logical terminal element
:PTE	Current physical terminal element
:TCE	Current task control element
:VECT	Vector table for debugger

Important: A single debug expression can reference only one system entity.

Address symbols and markers: Three special characters can be used in debugger expressions to address particular locations in a program or dialog:

Symbol	Symbol name	Designated location
@	At sign	Absolute address
\$	Dollar sign	Load address
¢	Cent sign	Address of current dialog process

Each type of location is described separately below.

Absolute address: The **at sign** (@) functions as the debugger marker that prefaces an absolute address notation. An absolute address cannot exceed eight digits.

Syntax for the marker is shown below:

▶▶ — @ hex-value —————▶▶

In a debug expression, @*hex-value* can be used interchangeably with the address notation *Xhex-value*. For example, an absolute address could be represented as @2B90 or X'002B90'; an offset value could be represented as +@C0 or +X'C0'.

▶▶ For more information about the hexadecimal values recognized by the debugger, see 2.7, “Data values” on page 2-17.

Load address: The **dollar sign** (\$) functions as the debugger label that expresses the load address of the current program. In a command that uses debug expressions, the dollar sign (\$) can be used by itself or in combination with other expression components.

This example illustrates the use of the dollar sign (\$) in an expression requesting a display of the current CSECT address:

```
list $
```

This example sets a breakpoint at an offset address 16 bytes from the load address:
at \$ + @10

Address of current dialog process: The **cent sign** (ϕ) functions as the debugger label that expresses the address of the current dialog process. In a command that uses debug expressions, the cent sign (ϕ) can be used by itself or in combination with other expression components.

This example illustrates the use of the cent sign (ϕ) to request the load address of the current dialog process:

```
list  $\phi$ 
```

2.2.2 User symbols

Additional work areas: User symbols identify storage areas set aside by the debugger as additional work areas. Each user symbol must be prefaced by a colon (:). The user symbols and their meanings are:

- **:DR n** for a debugger general register, where n represents the number of the register and can have a value of 0 through 15
- **:DREGS** for all debugger registers
- **:H1** and **:H2** for halfword 1 and halfword 2
- **:F1** and **:F2** for fullword 1 and fullword 2
- **:UCHR** for a 48-byte character area

You can also refer to specified sections of this area:

- **:UC0**, the first 16 bytes
- **:UC16**, the next 16 bytes
- **:UC32**, the last 16 bytes

Examples: The example below illustrates one way in which you can use the work areas as a debugging aid. In this example, when the program being debugged has reached a breakpoint and the debugger facility is in control, you can copy the current values in program registers to registers in the debugger work area. For instance, to save the contents of all 16 of the general registers of the program, issue this command:

```
set :dregs = :regs
```

To save the contents of a single register, copy the values currently in the user register to a debugger register, with a command in this format:

```
set :dr1 = :r1
```

Later in the debugger session, the user register previously saved can be restored with this command:

```
set :r1 = :dr1
```

Contents remain for session: You can modify or refer to the values in these registers at any time during a debugger session; debugger register contents remain only for the duration of the current session.

►► For more detailed information on the use of the SET command, see Chapter 3, “Debugger Commands” on page 3-1.

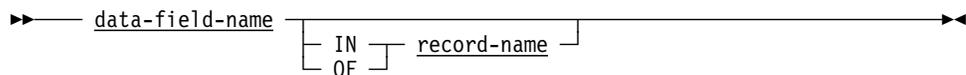
2.2.3 Program symbols

Data field names and line numbers are two types of program symbols used as components of debug expressions. Each of these components is discussed separately below, followed by a discussion of how program symbols can be qualified.

2.2.3.1 Data field names

When debugging a dialog during runtime, you can reference a specific data field.

Syntax: This is a summary of syntax for the use of data field names:



Parameters

data-field-name

Specifies the data field to be displayed. The name must be enclosed in quotation marks if it contains embedded delimiters. The data field name must be qualified if it is not unique to the process.

IN/OF record-name

Specifies the name of the record associated with the data field being requested. The record name must be enclosed in quotation marks if it contains embedded delimiters.

►► For a complete list of the delimiters used in debugger commands, see 2.6, “Delimiters” on page 2-16.

You cannot list or set data fields during the setup phase of a debugger session. If you try to, the debugger issues an error message, as in this example:

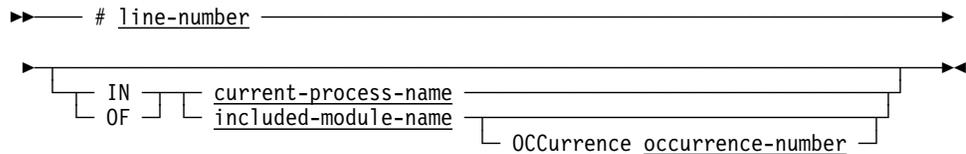
```
DEBUG >
list date

DC704900 LIST > DATE CANNOT BE RESOLVED
LIST DATE
DEBUG >
```

2.2.3.2 Line numbers

When debugging a dialog, you can use symbolic line numbers in a debug expression.

Syntax: This is a summary of syntax for the use of line numbers:



Parameters

#line-number

Specifies the process line number referenced in the expression. The line number can stand alone if it is unique to the current process.

current-process-name

Specifies what process currently being debugged contains the line number. The process name must be enclosed in quotation marks if it contains delimiters. The current process name is the default value.

included-module-name

Specifies the name of the included module called from the current process containing the line number. The name of the included module must be enclosed in quotation marks if it contains delimiters.

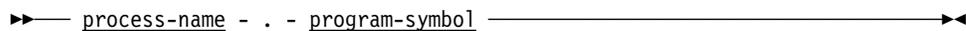
OCCurrence occurrence-number

Specifies the occurrence of the included module for modules included more than once in the process.

2.2.3.3 Qualifying program symbols

You can also use program symbols to refer to a line in another process without resetting the process currency.

Syntax: The syntax for temporary qualification is:



Parameters

process-name

Specifies the current process.

program-symbol

Specifies the program symbol used in this expression. The program symbol is a line number or a data field name. You can further qualify the symbol with the *included-module-name-qa* clause of a debug expression.

Example 1: Assume that the dialog being debugged has three processes: MIS-MAIN1 (the current process), MIS-MAIN2, and MIS-MAIN3. To set a breakpoint at line 200 in MIS-MAIN2, you can use the QUALIFY command to reset the currency to MIS-MAIN2 (QUALIFY PROCESS 'MIS-MAIN2' AT #200). However, to establish a breakpoint at line 200 *without* resetting currency, you can issue this command:

```
at 'mis-main2'.#200
```

Example 2: To set a breakpoint at line 150 in MIS-INC3, a module included by MIS-MAIN3, you can qualify the line number without changing currency from the MIS-MAIN1 process:

```
at 'mis-main3'.#150 of 'mis-inc3'
```

2.2.4 Expression operators

Standard operators: The standard operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Division

You can use these operators in any command containing a debug expression.

Special operators: The **percent sign (%)** is a special operator that you can use for **indirect addressing**. With indirect addressing, the address in the expression is not the address of the operand itself, but a pointer to a storage area that contains the address of the operand. When the percent sign precedes a valid debug expression, the content of the expression is used as the address of the target value.

Examples: Assume that register 3 contains the value BF040. You ask for display of the contents of register 3, like this:

```
list :r3
000BF040                                *.0.      *
```

In this example, the command points to the contents of register 3 as the target value for the display:

```
list %:r3
```

In response to the command above, the debugger locates the operand address (BF040) in register 3 and lists the contents stored at BF040:

```
000BF040 000047F0 C0280000 00000000 00000000 *.0.....*
```

Now you ask for display of the contents found at 10C010, the address supplied in the debug expression:

```
list reca +10
```

```
0010C010 000BF000 00000000 00000000 00000000 *..0.....*
```

In the next example, the relative storage location points to the address of the effective operand. The debugger responds by listing the contents of BF000, the operand address found at RECA+10:

```
list %(reca+10)
```

```
000BF000 D1D6C8D5 40E2D4C9 E3C80000 00000000 *JOHN SMITH.....*
```

2.3 Length attributes

The types of components used in an expression can determine the amount of information displayed or modified by the debugger in response to your request. When determining the length of a display, the debugger distinguishes between expressions with and expressions without associated data characteristics.

2.3.1 Expressions with data characteristics

When an expression component has associated data characteristics, the length of the display depends on:

- The length attribute of the symbol
- The length attribute of the end symbol
- The explicit length

Length attribute of the symbol: The length attribute of the symbol is used as the default value.

For example, this command requests the display of register 1:

```
list :r1
```

The length attribute of a general register is four bytes. The debugger uses the register attribute as the default value and issues the following display in response to the above command:

```
00000000
```

Length attribute of the end symbol: The length attribute of the end symbol in an expression range delineates the end of the display. For example, this command requests a display of register 1 through register 3:

```
list :r1 to :r3
```

The debugger responds with a display that includes the full four-byte length of register 3:

```
00000000 00000001 00000002
```

Explicit length: An explicit length overrides the display length implied by the data characteristics of a symbol.

This table lists the length attributes of debugger symbols:

Entity	Symbol	Length Attribute
Single registers	:Rn :SRn :DRn	4 bytes
Register blocks	:REGS :SREGS :DREGS	64 bytes
Program status word	:PSW	8 bytes
Halfwords	:H1 :H2	2 bytes
Fullwords	:F1 :F2	4 bytes
Line number	#Line-n	12 bytes
Control blocks	:BAT, :CSA, :DLB, :LTE, :PTE, :TCE, :VECT	Variable (depending on length of block)

2.3.2 Expressions without data characteristics

As soon as a component appears in an expression with any other component, it no longer has associated data characteristics. For example, :PTE is an expression with an implicit length attribute equal to the length of the control block, but :PTE +@10 is an expression without associated data characteristics.

Ways to determine length: When an expression component does not have associated data characteristics, the length of the display is based on:

- The default length of the command
- An explicit length
- The first byte of the end expression

Default command length: Default lengths vary for commands that use length parameters. For example, the default length is 16 bytes for the LIST command and 256 bytes for the SNAP command.

In this example, the display begins 32 bytes from the start of the current physical terminal element (PTE) for a length of 16 bytes:

```
list :pte +@20
```

Explicit length: You can supply an explicit length, which overrides the default length of the command. This example requests a 100-byte display that begins at the load address:

```
list $ 100
```

The next example requests that the display begin at an offset address for a length of 20 bytes:

```
list :pte +@10 len 20
```

First byte of the end expression: The first byte of the end expression in an expression range specifies the end of the display. For example, the debugger displays 17 bytes of memory in response to this command:

```
list @bf000 to @bf010
```

2.4 Parsing rules

Parameter order: The parameters of a command must appear **in the order specified in the syntax**.

In the display below, the first example is incorrect, because the BEFORE parameter cannot follow the AFTER parameter in an AT command:

```
at $ +@10 after 2 before 10 on ←incorrect order
```

```
at $ +@10 before 10 after 2 on ←correct order
```

Errors that stop execution: If one command in a string of debugger commands contains a **syntax error**, all following commands are parsed for syntax but **not executed**.

The command containing the syntax error may be partly executed. In the first example above, the part of the command preceding the error (*at \$ +@10 after 2*) will be executed:

```
DEBUG >
at $ +@10 after 2 before 10 on

AT $ +@10 ADDED
BEFORE 10 IGNORED
$
UNRECOGNIZABLE DEBUG COMMAND
DEBUG > AT $ +@10 AFTER 2 BEFORE 10 ON
DEBUG >
```

Commands that stop execution: If a RESUME, EXIT, IOUSER, MENU, PROMPT, or QUIT command is embedded in a string of concatenated debugger commands, **all successive commands in the string are ignored**.

2.5 Command modification

Rules of modification: Commands can be modified to specify different options or to turn off options completely. You can modify commands with expressions corresponding to the original command.

When you modify a command:

- A respecified option overrides its counterpart in the previous command
- All options specified in the previous command remain in effect unless overridden

Example: In this example these two commands

at \$ + 8 before 10 ignore

at \$ + 8 after 2 on

establish the breakpoint parameters specified in this display:

AT \$ + 8 BEFORE 10 AFTER 2 ON

2.6 Delimiters

Valid delimiters

*	Asterisk
	Blank
,	Comma
=	Equal sign
!	Exclamation point
-	Hyphen
%	Percent sign
.	Period
+	Plus sign
/	Slash

2.7 Data values

Valid data values: The debugger recognizes values supplied by the following types of numbers and strings:

Value	Description
Halfword values	Two-byte fixed-point values ranging from +32,767 to -32,768
Fullword values	Four-byte fixed-point values ranging from +2,147,483,647 to -2,147,483,648
Hexadecimal numbers	Values of one to eight hexadecimal digits preceded by an at (@) sign; can include characters A through F and numerals 0 through 9; when not used in a debug expression, contents must be paired hexadecimal digits
Decimal numbers	Values that can include decimal positions
Character strings	One- to 16-character alphanumeric values enclosed in single or double quotation marks and preceded by letter C (for example, C"F34"); can contain any printable character or blank
Hexadecimal strings	Even-numbered strings of up to 16 hexadecimal digits enclosed in single or double quotation marks and preceded by letter X (for example, X"C6F4"); paired characters A through F and paired numerals 0 through 9 for hexadecimal values
Numeric strings	Variable length numeric values enclosed in single or double quotation marks; preceded by letter H, F, or P to designate halfword values (H'0'), fullword values (F'555'), or packed decimal values (P"2315")

2.8 Command format

Rules

- One or more blanks must precede and follow all keywords
- Spaces are optional within an expression

An offset value can be expressed with separating blanks or without blanks. For example, the same command can be accurately formatted in any of these ways:

```
at @00bf280 + 10
at @00bf280+10
at @00bf280 +10
```

- The entire command string must not exceed twice the line length of the terminal
- Multiple commands can be entered on one prompt line

The commands can be separated with an exclamation point (!) delimiter, but the delimiter is not required. For example, the same command string can be accurately formatted in any of these ways:

```
DEBUG >
at $ + 8 every 5 on!resume
```

```
DEBUG >
at $ + 8 every 5 on resume
```

```
DEBUG >
at $ + 8 every 5 on
AT > $ + 8 ADDED
DEBUG >
resume
```

Chapter 3. Debugger Commands

- 3.1 Summary of commands 3-3
- 3.2 AT 3-4
- 3.3 DEBUG 3-7
- 3.4 EXIT 3-9
- 3.5 IOUSER 3-10
- 3.6 LIST 3-11
- 3.7 MENU 3-14
- 3.8 PROMPT 3-15
- 3.9 QUALIFY 3-16
- 3.10 QUIT 3-18
- 3.11 RESUME 3-19
- 3.12 SET 3-20
- 3.13 SNAP 3-24
- 3.14 WHERE 3-26

3.1 Summary of commands

This chapter presents a functional description, syntax, syntax rules and examples for each debugger command you can use during the setup or runtime phases. The commands are presented in alphabetical order.

This table summarizes the commands and their functions.

Command	Description
AT	Establishes or modifies breakpoints at specified locations in a user program
DEBUG	Designates an entity to be debugged or inquires about entities known to the debugger
EXIT	Returns control to the DC/UCF system, retaining the debugger control blocks created in the current session
IOUSER	Displays the screen current when a breakpoint, program check, or trapped abend is encountered
LIST	Displays session attributes, debugger variables, and areas of memory at your terminal
MENU	Invokes menu mode for a debugger session
PROMPT	Invokes prompt mode for a debugger session
QUALIFY	Assigns currency to a new process within the current dialog or inquires about program, dialog and process currencies in effect
QUIT	Terminates the debugger session and returns control to the DC/UCF system, clearing all control blocks created in the current debugger session
RESUME	Continues program or abend execution
SET	Allows you to modify storage and debugger session attributes
SNAP	Allows you to create and write a dump to the DC/UCF log
WHERE	Provides information about the last interrupt encountered in the entity being debugged

AFTer execution-count

Specifies an execution pause each time the same breakpoint instruction is encountered beyond *execution-count*.

EVERy 1

Causes the debugger to pause every time the breakpoint instruction is encountered. One is the default.

EVERy execution-count

Specifies an execution pause each time the counter for the specified breakpoint instruction reaches a multiple of *execution-count*.

ON

Sets a new breakpoint or resets the status of a breakpoint previously ignored. ON is the default in ADD format.

IGNore

Bypasses the specified breakpoint but increments the breakpoint counter.

OFF

Removes the breakpoint. Can be used only in INQUIRE format.

INQUIRE

Requests a listing of the breakpoint locations and characteristics. Can be used only in INQUIRE format.

Usage

Two formats: The AT command has two formats. The ADD format is used to set and modify breakpoints; the INQUIRE format is used to review breakpoint locations, if any have been set, as well as to modify the breakpoints.

Temporary processing halt: A breakpoint temporarily halts processing, allowing you to examine the results of execution up to the point of interruption. Processing is halted *before* the instruction at the breakpoint is executed. You can use the AT command in both the setup and the runtime phases of a debugger session.

Breakpoint count: In response to the INQUIRE format, the debugger displays all parameters in effect for the named breakpoints and indicates the breakpoint count. The breakpoint count (BKPT COUNT) shows how often the breakpoint has been encountered from the time the program received control via #LINK or #XCTL.

If you issue an AT INQUIRE command is issued during the setup phase, the breakpoint count documents the count from the most recently executed program. The breakpoint counter is reset to zero each time a #LINK or #XCTL is processed for the program.

Example 1: This command schedules program breaks on the second through ninth time the instruction at the address \$ + 8 is encountered.

```
DEBUG >  
at $ + 8 before 10 after 1
```

The debugger verifies the breakpoint with this message:

```
AT> $ + 8 ADDED
```

Once the breakpoint in the example above has been set, the debugger displays the following message in response to an AT \$ + 8 INQUIRE command:

```
AT> AT $ + 8 BEFORE 10 AFTER 1 EVERY 1 BKPT COUNT 0 ON
```

In this example, the default value is indicated for the EVERY parameter. BKPT COUNT 0 indicates that this breakpoint has not yet been encountered in the current execution of the program.

Example 2: When a breakpoint is reached during the runtime phase, the debugger displays a message that names the address, identifies the program, and displays the debug expression that established the breakpoint. For example, the following message would appear for a breakpoint established with an AT \$ + 8 command for program TESTPROG:

```
AT OFFSET @8 IN TESTPROG EXPRESSION $ + 8
```

Example 3: In CA-ADS dialogs you can set breakpoints by specifying a line number:

```
DEBUG >  
at #200
```

If line 200 is a valid address, the debugger responds to the above command as follows:

```
AT #200  
AT> #200 ADDED
```

Example 4: When debugging a dialog, you can set a breakpoint in a process other than the current process without changing the currency. In the following example where MIS-MAIN1 is the current process, a breakpoint is set at line 100 in a second process (MIS-MAIN2); MIS-MAIN1 retains its currency. As usual, the debugger sends a verifying message when the breakpoint address is valid.

```
DEBUG >  
at 'mis-main2'.#100  
  
AT 'MIS-MAIN2'.#100  
AT> 'MIS-MAIN2'.#100 ADDED  
DEBUG >
```

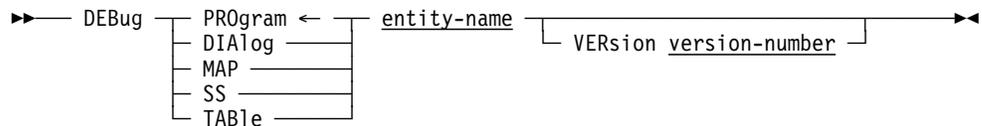
In the above example, the programmer encloses the process name in single quotation marks (') because the name contains an embedded hyphen (-). Quotation marks are required for any name that contains embedded delimiters.

3.3 DEBUG

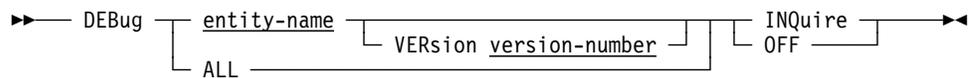
Purpose: Specifies the programs to be debugged or inquires about the debugged programs.

Syntax

ADD format



INQUIRE format



Parameters

PROgram/DIAlog/MAP/SS/TABle

Identifies the type of load module to be debugged. Used only in ADD format. PROGRAM is the default.

entity-name

Specifies the name of the entity to be used by the debugger as the current load module. *Entity-name* contains a maximum of eight characters.

ALL

Specifies all modules defined to the debugger during the current session. Can be used only in INQUIRE format.

VERSION version-number

Identifies the version of the program being debugged.

If the version is *not* specified:

- In ADD format, the debugger uses the version set with DCUF TEST, or version 1 if DCUF TEST hasn't been issued
- In INQUIRE format, the debugger displays all versions if none is specified

INquire

Requests a listing of the modules being debugged in this session.

OFF

Terminates all debugging for the specified programs for the remainder of the session.

Usage

Functions of DEBUG: The word DEBUG has several functions:

- **Task code** used to initiate a debugging session
- **Prompt** displayed during a debugging session in prompt mode
- **Command** used during the setup phase to designate the programs to be debugged or to inquire about the debugged programs

You can use the DEBUG command only during the setup phase.

Special copy loaded: When you issue the DEBUG command for a module, a special copy is loaded, so that setting breakpoints and making data changes will not affect other users.

Two formats: The DEBUG command has two formats. The ADD format initially identifies the entities to be debugged; the INQUIRE format lists entities defined to the debugger in a given session.

Example 1: This example illustrates the use of the DEBUG task code in conjunction with the DEBUG command to transfer control from DC/UCF to the debugger and to define a module to the debugger; the debugger verifies the commands and displays the DEBUG> prompt in response:

```
ENTER NEXT TASK CODE:
debug debug testprog

DEBUG TESTPROG
DEBUG > DEBUGGING INITIATED FOR TESTPROG VERSION 1
DEBUG >
```

Example 2: In this example, the DEBUG command names the load module to be debugged:

```
DEBUG >
debug dialog msgtext version 3

DEBUG DIALOG MSGTEXT VERSION 3
DEBUG > DEBUGGING INITIATED FOR MSGTEXT VERSION 3
DEBUG >
```

Example 3: This command requests a list of all programs defined to the debugger during the current session:

```
DEBUG >
debug all inquire

DEBUG ALL INQUIRE
PROGRAM TESTPROG VERSION 1
DIALOG MSGTEXT VERSION 3 PROCESS MSG-MAIN1 CURRENT
DEBUG >
```

3.4 EXIT

Purpose: Returns control to DC/UCF and retains the debugger control blocks.

Syntax

▶— EXIT —————▶

Usage: Use EXIT to complete the setup phase and return to DC/UCF.

In a concatenated list of commands, the debugger ignores any command that follows the EXIT command.

Important: In debugging a dialog, the EXIT command causes rollbacks to be issued for both the database, if a run unit is open, and the task.

Examples: This example illustrates the use of the EXIT command and the resulting system response:

```
DEBUG >  
exit
```

```
EXIT  
EXIT DEBUGGER  
ENTER NEXT TASK CODE:
```

3.5 IOUSER

Purpose: Redisplays the screen that appeared at your terminal immediately before the debugger processed the breakpoint or trappedabend.

Syntax

▶— IOUser —————▶

Usage: After the screen is redisplayed, you can return to the menu mode screen or to the DEBUG> prompt by pressing any control key.

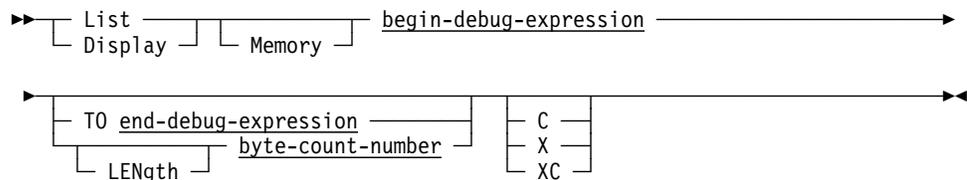
You can issue the IOUSER command only at runtime. In a concatenated list of commands, the debugger ignores any command that follows the IOUSER command.

3.6 LIST

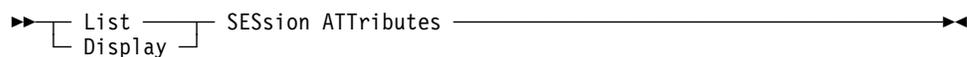
Purpose: Displays selected areas of storage and session attributes at your terminal.

Syntax

MEMORY format



ATTRIBUTES format



Parameters

begin-debug-expression

Specifies the beginning location of the display. *Begin-debug-expression* can include multiple debug expressions and it resolves to an address for which you have retrieval security.

►► For information on the security methods used by the debugger, see *CA-IDMS Security Administration*.

►► For more information on the values used in a debug expression, see 2.2, "Expression components" on page 2-4 in Chapter 2, "Command Considerations."

end-debug-expression

Specifies the ending location of the display. *End-debug-expression* can include the same debugger entities as those specified in *begin-debug-expression*. The expression must resolve to a valid address for which you have retrieval security.

byte-count-number

Indicates the number of bytes to be displayed.

Important: If a resource is listed and the length or ending address exceeds the resource boundary, the list is truncated at the boundary, and the debugger issues a warning message.

C

Requests a display in character format.

X

Requests a display in hexadecimal format.

XC

Requests a display in both hexadecimal and character format.

Usage

Two formats: There are two formats for the LIST command. The MEMORY format requests a display of the contents of memory; the ATTRIBUTES format requests a display of session attributes.

Rules for default length: When neither *end-debug-expression* nor *byte-count-number* is specified, the default length is based on these rules:

- If the expression is composed of a single symbol, the data characteristics of the symbol determine the default length. The number of bytes displayed is equal to the default length of the symbol.
- If the expression does not have data characteristics, the default length is 16 bytes.

Format specified for this command: **XC/X/C** specifies the format for the requested information. This specification can override the type of display previously established as a session attribute; the override is only valid for the duration of this command. See the ATTRIBUTES format of the SET command to reestablish the session attributes more permanently.

Example 1: This command requests a list of the storage contents beginning at @BF002, for a length of 48 bytes:

```
list @bf002 48
```

The debugger responds with a display of the beginning address and the requested storage contents:

```
000BF002      47F0 C028....  *...0.....*
000BF010  58509002 .....  *.....*
000BF020  4780C12A .....  *..A.....*
000BF030  4770          *..          *
```

The first line of the storage display is indented for a space of two bytes, reflecting the exact beginning address.

Example 2: This command instructs the debugger to display the physical terminal element (PTE) control block from the beginning to the end of the entity. The length of the data field is determined by the data characteristics of the PTE.

```
list :pte
```

Example 3: The next command instructs the debugger to display storage contents beginning at @BF020. Since this expression has no data characteristics, the display defaults to 16 bytes.

```
list @bf020
```

Example 4: In debugging CA-ADS dialogs you can use a data field name:

```
list date
```

The debugger responds by displaying the requested information:

```
001C2C50 F8F4F0F3 F0F1          *840301          *
```

Important: You cannot refer to data fields of Assembler, COBOL, or PL/I programs by name.

Example 5: You can also use a line number:

```
list #100
```

Example 6: When field names or line numbers are not unique, you must qualify them. This example lists line 100 from a process other than the current dialog process:

```
list 'process-b'.#100
```

Example 7: This example qualifies a request by specifying the display of a field name *USERID-1301* from a record *EMPLOYEE-1301*:

```
list 'userid-1301' in 'employee-1301'
```

Example 8: This is an example of the ATTRIBUTES format:

```
DEBUG >
list session attributes

LIST SESSION ATTRIBUTES
LIST > SESSION ATTRIBUTES
      LIST:          CHAR
      TEST VERSION: 2
DEBUG >
```

This display indicates that DCUF TEST 2 and SET CHAR were issued.

3.7 MENU

Purpose: Switches the debugger session from prompt mode to menu mode.

Syntax

►► MENU screen-name ◀◀

Parameter

screen-name

Indicates the name of a global help screen or an activity screen to be displayed. If *screen-name* is not specified, the debugger displays the Usage screen, the top-level global help screen that presents a list of debugger commands and functions.

Usage: The MENU command is executed in prompt mode and switches the debugger session from prompt mode to menu mode. MENU is disabled in menu mode.

In a concatenated list of commands, the debugger ignores any command that follows the MENU command.

Example: This command instructs the debugger to switch from prompt mode to menu mode with the display of the activity screen for the LIST command:

```
DEBUG >  
menu list
```

►► For a complete discussion of the screens available in menu mode, see Chapter 4, “Debugging in Menu Mode” on page 4-1.

3.8 PROMPT

Purpose: Switches the debugger session from menu mode to prompt mode.

Syntax

►► — PROMpt ————— ◄◄

Usage: The PROMPT command is executed in menu mode and switches the debugger session from menu mode to prompt mode. PROMPT is disabled in prompt mode.

In a concatenated list of commands, the debugger ignores any command that follows the PROMPT command.

3.9 QUALIFY

Purpose: Establishes a new current process or inquires about the current program, or dialog and process.

Syntax

RESET format

```

▶▶ QUALify ┌──────────────────────────┐ PROCess process-name ───────────▶
           │ ┌──────────┐             │
           │ │ DIAlog dialog-name │ │
           └──────────────────────────┘
           ┌──────────────────────────┘
           │ ┌──────────┐             │
           │ │ VERsion version-number │ │
           └──────────────────────────┘

```

INQUIRE format

```

▶▶ QUALify INquire ───────────────────────────────────────────────────────────▶

```

Parameters

DIAlog dialog-name

Specifies the dialog currently defined to the debugger. Only current dialog can be qualified.

PROCCess process-name

Specifies the new dialog process to become current. Enclose the process name in single quotation marks if the name contains embedded delimiters.

VERsion version-number

Specifies the version number of the current dialog.

Usage

Resetting currency: When a dialog is defined to the debugger, the premap process becomes the current process by default. You can use the QUALIFY command to assign currency to a different process within the current dialog.

Two formats: The QUALIFY command has two formats. The RESET format resets currency; the INQUIRE format requests a display of the current program, or the current dialog and process.

The QUALIFY command can be used in both the setup and the runtime phases of a debugger session.

Example 1: You can inquire about the current dialog process:

```
DEBUG >  
qualify inquire
```

The debugger responds in this format:

```
QUALIFY INQUIRE  
DIALOG MISINDC VERSION 1 PROCESS MIS-MAIN1 CURRENT  
DEBUG >
```

Example 2: These commands reassign currency to MIS-MAIN2 and set a breakpoint at line 200 within MIS-MAIN2:

```
qualify proc 'mis-main2' at #200
```

The debugger responds like this:

```
QUALIFY PROCESS 'MIS-MAIN2'  
QUALIFY > CURRENCY SET  
AT #200  
AT > #200 ADDED  
DEBUG >
```

3.10 QUIT

Purpose: Terminates a debugger session and returns control to DC/UCF, clearing the debugger control blocks.

Syntax

►— QUIT —————►

Usage: The QUIT command discontinues debugging and lets you enter a new task code in response to the Enter Next Task Code prompt.

In a concatenated list of commands, the debugger ignores any commands that follow the QUIT command.

Important: In debugging a dialog, the QUIT command causes rollbacks to be issued for both the database, if a run unit is open, and the task.

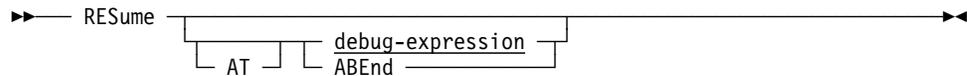
Example: This is how the system responds to the QUIT command:

```
DEBUG >  
quit  
QUIT  
QUIT DEBUGGER  
ENTER NEXT TASK CODE:
```

3.11 RESUME

Purpose: Instructs the runtime system to continue program execution at the next instruction or a specified location or to continue standard processing of an abend.

Syntax



Parameters

debug-expression

Specifies the location at which execution is to continue, if other than the instruction immediately following the breakpoint. *Debug-expression* can include multiple debug expressions, and it resolves to an address containing a valid instruction or a valid CME (CA-ADS dialogs only).

►► For more information about the values used in a debug expression, see 2.2, “Expression components” on page 2-4 in Chapter 2, “Command Considerations.”

ABEnd

Specifies that standard DC/UCF abend processing, including the execution of any STAE set, should continue.

Usage: You can issue the RESUME command only at runtime.

When program execution resumes at an address other than the address of the instruction immediately following the breakpoint, you must be sure that the program environment (for example, the contents of registers and storage) is appropriate for running the program.

Examples: This command requests that execution of the program resume with the instruction at the breakpoint:

```
resume
```

This command requests that program execution resume at the load address:

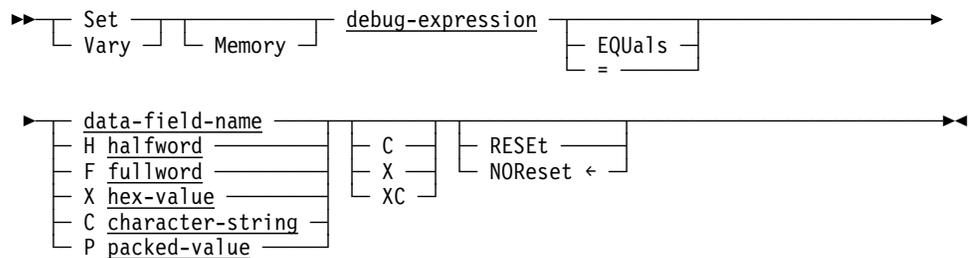
```
resume $
```

3.12 SET

Purpose: Modifies selected areas of storage and debugger symbols.

Syntax

MEMORY format



ATTRIBUTES format



Parameters

debug-expression

Specifies the beginning location of the entity to be modified. *Debug-expression* can include multiple debug expressions, and it resolves to an address for which you have update security.

►► For information on the security methods used by the debugger, see *CA-IDMS Security Administration*.

►► For more information on the values used in a debug expression, see 2.2, "Expression components" on page 2-4 in Chapter 2, "Command Considerations."

data-field-name

Identifies a specific data field value. Can be used in CA-ADS dialogs only.

►► For a complete discussion of the use of field names, see 2.2.3, "Program symbols" on page 2-7 in Chapter 2, "Command Considerations."

Hhalfword

Is a halfword number. H specifies the halfword format; *halfword* represents the actual data content and must be enclosed in single quotation marks.

Ffullword

Is a fullword number. F specifies the fullword format; *fullword* represents the actual data content and must be enclosed in single quotation marks.

Xhex-value

Is a hexadecimal string. X specifies the hexadecimal format; *hex-value* represents the actual data content and must be enclosed in single quotation marks.

Ccharacter-string

Is a character literal used to assign alphanumeric or symbolic character values. **C** specifies the character format; *character-string* represents the actual data content and must be enclosed in single quotation marks.

Ppacked-value

Is an assigned packed decimal value. **P** specifies the packed decimal format; *packed-value* represents the actual data content and must be enclosed in single quotation marks.

C

Requests a display in character format.

X

Requests a display in hexadecimal format.

XC

Requests a display in both hexadecimal and character format.

RESEt

Specifies that the named storage be reset to its original value at the end of the debugging session. This option is not supported for release 10.2 of the debugger.

NOReset

Specifies that the storage is not to be reset to its original value at the end of the debugging session. This option does not affect storage in the debugged program itself since a special copy of the program is loaded for the debugging session. NORESET is the default.

CHAR

Requests a display in character format for ATTRIBUTES format.

HEX

Requests a display in hexadecimal format for ATTRIBUTES format.

BOTH

Requests a display in both hexadecimal and character format for ATTRIBUTES format.

Usage

Two formats: The SET command has two formats. The MEMORY format specifies the values assigned to a given debug expression; the ATTRIBUTES format specifies the debugger session attributes to be established.

When *debug expression* is a symbol with data characteristics (for example, :REGS), the length of the symbol is used in the set. When the expression does not have data characteristics (for example, \$ + 10), the data characteristics of the source field are used in the set.

Important: The debugger does not allow a set across resource boundaries.

Character and hexadecimal format: **C/X/XC** in the MEMORY format specifies how the information is to be listed. This specification can override the session attributes previously established for the session; the override is valid only for the duration of this command. To reestablish the session attributes more permanently use the ATTRIBUTES format.

Example 1: This command modifies the contents of a program register:

```
DEBUG >
set :r7 x'00000001' x
```

The debugger responds to the X parameter with the hexadecimal display of the original value and the reset value:

```
SET :R7 X'00000001' X
OLD
00000000
NEW
00000001
DEBUG >
```

Example 2: This command modifies storage at an offset address:

```
DEBUG >
set $ + 8 = x'58' x
```

The debugger responds:

```
SET $ + 8 = X'58' X
OLD
000BF008 41
NEW
000BF008 58
DEBUG >
```

Example 3: This command modifies storage at the same address with a fullword value:

```
DEBUG >
set $ + 8 equ f'58' x
```

The debugger responds:

```
SET $ + 8 EQU F'58' X
OLD
000BF008 4130C050
NEW
000BF008 0000003A
DEBUG >
```

Example 4: This is an example of the ATTRIBUTES format:

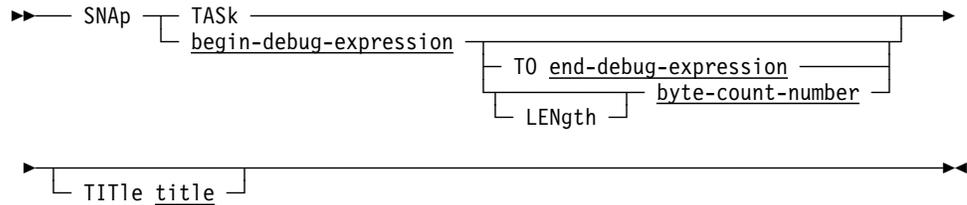
```
DEBUG >  
set char
```

```
SET CHAR  
SET ATTRIBUTE CHAR  
DEBUG >
```

3.13 SNAP

Purpose: Allows you to create a dump and write it to the DC/UCF log.

Syntax



Parameters

TASK

Requests a dump of all resources associated with the executing task, as well as the Task Control Element (TCE) and the Dispatch Control Element (DCE).

begin-debug-expression

Specifies the location at which to begin the snap. *Begin-debug-expression* can include multiple debug expressions, and it resolves to an address for which you have retrieval security.

►► For information on the security methods used by the debugger, see *CA-IDMS Security Administration*.

►► For more information on the values used in a debug expression, see 2.2, "Expression components" on page 2-4 in Chapter 2, "Command Considerations."

end-debug-expression

Specifies the ending location of the display. *End-debug-expression* can include the same debugger entities as those specified in *begin-debug-expression*. The expression must resolve to a valid address for which you have retrieval security.

byte-count-number

Specifies the number of bytes to be displayed.

TITLE title

Specifies an optional title for the snap. The title must be enclosed in single quotation marks ('), may not exceed 32 characters, and must be prefaced by a valid ASA carriage control character. These are the valid carriage control characters:

(Space bar)	Space one line
0	Space two lines
-	Space three lines
1	Skip to the top of the next page

The length of 32 characters includes the carriage control character. Code apostrophes in the title as two single quotation marks ("). They are counted as one character position.

When a title is not specified, a default title is written to the log.

Usage

Types and timing: You can use a SNAP command for a Task snap or a snap of specific area; the command is valid at any point in a debugger session.

You can examine the Snap dumps online with OLP (OnLine Plog), or make a hard copy by running the print log functions of the Batch Command Facility utility.

►► For more information see *CA-IDMS Utilities*.

Default length: When neither *end-debug-expression* nor *byte-count-n* is specified, the default length is based on these rules:

- If the expression is composed of a single symbol, the data characteristics of the symbol determine the default length. The number of bytes dumped is equal to the default length of the symbol.
- If the expression does not have data characteristics, the default length is 256 bytes.

Example 1: This command causes a snap to begin at the load address and terminate at @000BF050; the default title is to be used:

```
DEBUG >
snap $ to @bf050
```

The default title takes the form:

```
SNAP command-entered USER user-id
```

For example, if the user ID is MMC, the default title is:

```
SNAP $ TO @000BF050 USER MMC
```

Example 2: This command requests a snap starting at the load address for 256 bytes; the default title is to be used:

```
DEBUG >
snap $
```

Example 3: This command requests a task snap; the title IDNSTEST, positioned at the top of a display page, will be used for the dump:

```
DEBUG >
snap task title 'lidmstest'
```

3.14 WHERE

Purpose: Provides information about the last interrupt of the entity being debugged.

Syntax

▶▶— WHEre —————▶▶

Usage: You can issue the WHERE command only at runtime.

Example: This is how the debugger responds to the WHERE command:

```
DEBUG >  
where
```

```
WHERE > @000BF010 LAST INTERRUPT MESSAGE FOLLOWS  
AT OFFSET @10 IN TSTPROG EXPRESSION $ + @10
```

Chapter 4. Debugging in Menu Mode

- 4.1 Features of menu mode 4-3
- 4.2 Screen design 4-4
 - 4.2.1 Heading area 4-4
 - 4.2.2 Display area 4-7
 - 4.2.3 Specification area 4-7
 - 4.2.4 Selection area 4-7
- 4.3 Accessing screens 4-9
 - 4.3.1 Screen hierarchy 4-9
 - 4.3.2 Screen sequence 4-10
 - 4.3.3 Selection processing 4-10
 - 4.3.4 Command currency 4-11
- 4.4 Activity screens 4-12
 - 4.4.1 At screen 4-12
 - 4.4.2 Debug screen 4-13
 - 4.4.3 List screen 4-14
 - 4.4.4 Resume screen 4-15
 - 4.4.5 Set screen 4-16
 - 4.4.6 Snap screen 4-17
- 4.5 Global help screens 4-19
 - 4.5.1 Usage screen 4-19
 - 4.5.2 Symbols screen 4-19
 - 4.5.3 Keys screen 4-20

4.1 Features of menu mode

Menu mode facilities: Menu mode provides screens that allow you to choose any of the debugging activities that can be performed in prompt mode. Fixed-format activity screens are available for each command to simplify the process of debugging. Menu mode also offers several help facilities.

Chapter contents: This chapter discusses the following features of menu mode:

- Screen design — Standard format of the activity and help screens
- Accessing screens — Moving between screens
- Activity screens — Descriptions of the variable fields on the command-specific activity screens
- Global help screens — Descriptions of the global help screens

4.2 Screen design

Screen areas: The menu mode screens are designed for ease of use. Each screen has a:

- Heading area
- Display area
- Specification area
- Selection area

This diagram shows the areas of the screen:

```

IDMS-DC REL 15.0 ONLINE DEBUGGER *** LIST ***          SETUP   PAGE 1 OF 1
PROGRAM:          V:          CSECT:
->

LIST: M (M-MEMORY/A-ATTRIBUTES)
MEMORY ONLY:
  BEGIN LIST AT:

  LENGTH.....:          - OR -   END LIST AT:

  LIST FORMAT..: B (C-CHARACTER/X-HEX/B-BOTH)

NEXT _ ACTIVITY OR _ HELP:
  _ AT      _ LIST      _ SET      _ SNAP      _ RESUME      _ DEBUG      _ WHERE

HELP _ EXIT      _ PROMPT      _ QUIT      _ IOUSER
SCREENS: _ USAGE      _ SYMBOLS      _ KEYS

```

Each of the screen areas is described below.

4.2.1 Heading area

Contents: The heading area includes three lines:

- Header line
- Currency line
- Prompt line

Header line: The header line contains several fields:

- The **PF-key** field provides a two-position entry area for simulation of a program-function key. For example, typing a 5 in this field and pressing [Enter] has the same effect as pressing [PF5].

The simulated PF-key field is useful when your terminal does not have program-function keys. You can specify the numerals 1 through 24, as well as EN for [Enter], CL for [Clear], P1 for [PA1], and P2 for [PA2].

- **Product name** and **release number** fields supply information formatted like this:
IDMS-DC REL n.n ONLINE DEBUGGER
- The **screen label** field indicates the name of the current screen. The screen name changes as you move from one activity or help screen to another in the debugging process. (The sample screen is the List screen.)
- The **session mode** field indicates whether you are in the setup or runtime phase of the debugging process. (The sample screen indicates a setup phase.)
- **Page notations** supply the current page and the total number of pages available for the given display. The sample screen indicates that you are viewing the first page of a one-page display. Typically the help screens have more than one page. You can display a different page by:
 - **Overwriting** the current page number on the header line and pressing [Enter]
 - **Using the designated control key** to scroll backward or forward.

Default control key assignments This table presents a list of the default control key assignments for the debugger:

Key	Action	Description	Function
[PF1]	Usage	Displays the Usage screen	2
[PF2]	Unassigned		5
[PF3]	Activity	Displays the activity screen for the current command	3
[PF4]	Help	Displays the help screen for the current command	4
[PF5]	Symbols	Displays the Symbols screen	9
[PF6]	Keys	Displays the default control key assignments	6
[PF7]	Scroll up	Displays the previous page	7
[PF8]	Scroll down	Displays the next page	8
[PF9]	Prompt	Returns the debugger to prompt mode	1
[PF10]	Unassigned		15
[PF11]	Unassigned		11
[PF12]	Reserved		12
[PA1]	Refresh	Refreshes the current screen	14

Key	Action	Description	Function
[PA2]	Exit	Exits the debugger	10
[Clear]	Return	Goes back one level	16
[Enter]	Process	Processes the current screen	13

The default control key assignments can be changed at DC/UCF system generation time with the KEYS statement.

►► For more information on the KEYS statement used in system generation, see *CA-IDMS System Generation*.

The Keys screen displays the key assignments for your particular installation.

Currency line: The currency line displays the **current values for five variable fields**:

- The **entity type** indicates whether a program, dialog, map, table or subschema load module is currently being debugged
- The **entity name** field displays the name of the current entity
- **V:version-n** displays the version number associated with the current entity
- The **section type** field indicates whether a dialog process or a program CSECT is currently being debugged.
- The **section name** field displays the current CSECT or process name

When the current entity is a **program**, the currency line reads like this:

```
PROGRAM: PROG01 V:3 CSECT:
```

When the current entity is a **dialog**, the currency line reads like this:

```
DIALOG: MISINDC V: 1 PROCESS: MIS-MAIN2
```

The currency line remains constant until there is a change in the entity of the CSECT or process being debugged. You can change the current CSECT or process by:

- Overwriting the name on the screen and pressing [Enter] to automatically initiate the QUALIFY command
- Issuing the QUALIFY command on the prompt line

Prompt line: The prompt line is prefaced by an arrow (→) and functions in the same manner as the DEBUG> prompt in prompt mode. You can use the prompt line on any screen during menu mode; you can submit a single debugger command or a string of commands at any time.

►► For a complete discussion of the debug expressions and commands that you can enter on the prompt line, see 2.2, “Expression components” on page 2-4 in Chapter 2, “Command Considerations” and Chapter 3, “Debugger Commands” on page 3-1.

4.2.2 Display area

Contents: The display area is reserved to display:

- The information being presented for each of the help screens
- Output you have requested from the debugger
- Informational and error messages supplied by the debugger

4.2.3 Specification area

Contents: The specification area contains fields in which you can specify the desired options for the command being used. The contents of the specification area vary from screen to screen, and not all screens have a specification area.

Screen content in the specification area of the activity screens is saved for as long as the command is current. This feature allows you to suspend action on a partially filled screen while seeking further information.

For example, you can:

- Begin to fill the activity screen for the List command
- Switch to the Symbols help screen to review program or debugger symbols
- Return to the List screen, where all previous input remains intact

►► For more information about command currency, see 4.3.4, “Command currency” on page 4-11.

4.2.4 Selection area

List of procedures: The selection area presents a list of the debugger commands and global help screens that you can initiate from the screen. You can select the next action by entering any character other than a blank or an underscore in the response field to the left of an activity or help function.

Two sections: You can select actions from one of two sections:

- Section A displays the choice of command-specific activity and help screens:

```
NEXT  _  ACTIVITY OR  _  HELP
      _  AT      _LIST  _SET   _SNAP  _RESUME  _DEBUG  _WHERE
      _EXIT    _PROMPT  _QUIT  _IOUSER
```

- Section B displays the choice of global help screens:

HELP SCREENS: _ USAGE _ SYMBOLS _ KEYS

Command-specific activities: When choosing from Section A, you first select Activity (the default) or Help and then choose one of the commands. If you select Activity, the system can:

- Execute immediately an EXIT, PROMPT, QUIT, or IOUSER command
- Display the activity screen for an AT, LIST, SET, SNAP, RESUME, or DEBUG command
- Display the information requested by the WHERE command

Control keys can also be used to request activities.

Selecting help If you select Help from Section A, the system displays a command-specific help screen.: If you mark the select byte for Activity or Help but do not choose a specific command, the system displays the activity or help screen for the current command. The debugger system displays an error message if there is no current command.

You can choose a global help screen from Section B.

Each of the activity screens and global help screens is described in detail later in this chapter.

4.3 Accessing screens

Considerations: When moving between screens, you need to consider:

- Screen hierarchy
- Screen sequence
- Selection processing
- Command currency

4.3.1 Screen hierarchy

Three screen levels: The debugger supports three levels of screens:

Screens	Level
Usage screen	Top
Activity screens	Second
Help screens	Third

Usage screen: The Usage screen is an informational global help screen that contains a list of the debugger commands and a brief description of their functions. The Usage screen is the default screen for the MENU command.

Activity screens: Activity screens are screens that provide you with an area for specifying command options. The debugger provides activity screens for the AT, DEBUG, LIST, RESUME, SET, and SNAP commands. You can initiate these commands from the activity screens once you've entered the necessary information in the specification area.

Help screens: Help screens provide two types of assistance:

- Command-specific help screens supply tutorial information on all the debugger commands. When the command is one that uses an activity screen, the help screen for that command also describes the field options.
- Global help screens provide information not associated with a particular command. For example, the Symbols screen enables you to choose a display of program and debugger symbols for the current session, and the Keys screen displays site-specific PF-key assignments.

4.3.2 Screen sequence

Next activity or [Clear]: You can change to the next screen by:

- Explicitly specifying the next activity to be performed
- Using the [Clear] key (or the key associated with function 16)

Specifying the next activity: You can select an activity by:

- **Using the control key** associated with the activity to be performed

Default control key assignments are discussed in "Heading area" earlier in this chapter. The Keys screen displays a list of the current function assignments for your installation.

- **Entering a nonblank character in the response field** to the left of the activity to be performed

You can use any character other than a blank or an underscore. The choice of actions is listed in the selection area of each screen. For a description of the selection area, see 4.2, "Screen design" on page 4-4 earlier in this chapter.

Using [Clear]: The performance of [Clear] depends on the screen level from which you initiate the action:

- From an activity screen, [Clear] displays the Usage screen
- From the Symbols screen, the Keys screen, or one of the command-specific Help screens:
 - When there is a **current command**, [Clear] displays the activity screen for the current command
 - When there is **no current command**, [Clear] displays the Usage screen
- From the Usage screen, [Clear] returns control to DC/UCF

4.3.3 Selection processing

Order of precedence: The debugger determines its next action based on these factors, in order of precedence:

1. The **control key** used to initiate a particular action
2. The **select byte(s)** marked in the selection area
3. The **page number** designated in the heading area
4. Commands initiated from the **menu-mode prompt line**
5. Commands initiated from the **specification area**

Once an action is identified for processing, the system ignores all other requested actions.

Example: For example, if the USAGE screen is your current screen and you choose the AT activity from the selection area and then press the CLEAR key, the CLEAR key takes precedence and you are returned to DC/UCF.

4.3.4 Command currency

Repeating a command: Command currency is a feature of menu mode that simplifies the debugging process when you use the same command in successive actions. With command currency, you select the command the first time only.

Defining the current command: The current command is defined as the most recent debugger command referenced on a command-specific help screen or an activity screen. No current command exists until you take either of two actions:

- Use the *screen-name* option with the MENU command to name an activity screen.
For example, the command MENU LIST establishes the LIST command as the current command.
- Designate a command from the activity or help selection list at the bottom of any screen.

The newly-selected command functions as the current command.

The current command is the default command. This means that the debugger system automatically displays the appropriate screen for the current command.

You can choose Activity or Help in the selection area, or press the control key associated with either of these actions, without specifying a command. If no current command has been established when you make any of the above choices, the debugger system displays an error message.

Changing command currency: You can change command currency in the same way you establish it.

For example, if the current command is LIST, mark the select byte for the SET command and press the control key associated with the current command-specific help screen (for example, [PF4]). The Help screen for SET appears, because SET is the newly-designated current command.

Command currency does *not* change when you:

- Enter a command on the screen prompt line.
For example, while setting breakpoints with the At screen, you can use the prompt line to request a memory display with the LIST command. In this case, the AT command remains as the current command.
- Select a global help screen, that is, a screen that is not associated with a specific debugger command.

For example, you can move from the LIST command activity screen to the Usage, Symbols, or Keys screen without changing command currency.

4.4 Activity screens

Format: An activity screen is provided for any debugger command that has fields for user-supplied values. Some fields are required and others have default values or are optional. The command-specific area of the activity screens is the **specification area**; all other areas have the standard format presented in "Screen design" above.

4.4.1 At screen

Purpose: You can use the At screen to:

- Add breakpoints
- Modify breakpoints
- Delete breakpoints
- Inquire about the breakpoints that have already been set

As explained in "Debugger features", Chapter 1, breakpoint temporarily halts processing, allowing you to examine the results of execution up to the point of interruption.

Remember: Processing is halted before the instruction at the breakpoint is executed.

The AT command can be used in both the setup and runtime phases of the debugger.

Two sections: The specification area of the At screen has two separate sections:

- The first section sets new breakpoints:
ADD BREAKPOINT AT:
BEFORE: MAX AFTER: 0 EVERY: 1
- The second section inquires about existing breakpoints, or deletes them:
OTHER ACTION.....: (I-INQUIRE/D-DELETE/G-IGNORE)
BREAKPOINT OR <ALL>:
- Both sections modify breakpoints

You can specify both sections of the screen at the same time.

Field options: These are the field options for this area:

ADD BREAKPOINT AT:

Designates the location in your program that will contain a breakpoint. The specified value can include one or more debug expressions resolving to an address that contains a valid instruction or, for CA-ADS dialogs, a valid CME.

Remember: It is not valid to set a breakpoint at the target of an Assembler execute (EX) instruction.

BEFORE: MAX

Specifies the execution pause on encountering the instruction up to, but not including, the specified number of times. The default (MAX) is to pause as many times as the instruction is encountered.

AFTER: 0

Specifies that the debugger will pause at the breakpoint after the instruction has been executed the specified number of times. The default (0) is to start pausing when the instruction is first encountered.

EVERY: 1

Specifies an execution pause every time the counter for the breakpoint instruction reaches a multiple of the value specified. The default (1) is to pause every time the instruction is encountered.

Tip: If you don't change the defaults, the debugger will pause each time the breakpoint instruction is encountered.

OTHER ACTION...: (I-INQUIRE/D-DELETE/G-IGNORE)

- I requests a listing of the breakpoint location and characteristics
- D removes the breakpoint
- G bypasses the breakpoint but increments the breakpoint counter

BREAKPOINT OR <ALL>:

Indicates the breakpoints affected by the Other Action field. You can indicate a specific breakpoint (that is, a *debug-expression*), or specify that the action applies to ALL breakpoints within the current program or dialog.

4.4.2 Debug screen

Two sections: The specification area of the Debug screen also has two sections:

- The first section designates the load module to be debugged:

```
DEBUG LOAD MODULE...:   TYPE: P (P-PGM/D-DIALOG/M-MAP/T-TABLE/S-SS)
VERSION.....:
```

- The second section inquires about certain debugged modules or removes modules from the debugging process:

```
OTHER ACTION.....:   (I-INQUIRE/D-DELETE)
LOAD MODULE OR <ALL>:
VERSION.....:
```

You can submit both types of requests at the same time.

Field options: These are the field options for this area:

DEBUG LOAD MODULE...:

Identifies the name of the entity to be debugged. The entity name can be up to eight characters long.

TYPE: P (P-PGM/D-DIALOG/M-MAP/T-TABLE/S-SS)

Identifies the type of module to be debugged:

- P (the default) identifies a program
- D identifies a CA-ADS dialog
- M identifies a map
- T identifies an edit or code table
- S identifies a subschema

VERSION.....:

Identifies the version of the load module to be debugged. If the version is not specified, the debugger uses the version you have set with DCUF TEST, or if none, version 1.

OTHER ACTION.....: (I-INQUIRE/D-DELETE)

- I requests a display of the load module(s) being debugged in this session
- D requests that the specified module(s) be removed from the list of load modules known to the debugger

LOAD MODULE OR <ALL>:

Indicates the load module(s) affected by the specified Other Action value. An *entity-name* identifies the single load module for which **I** or **D** is requested. Using **All** requests **I** or **D** for all load modules being debugged.

VERSION.....:

Identifies the version of the load module for which **I** or **D** is requested. If no version is specified and there is more than one version of the load module being debugged, the debugger displays or deletes all versions. If a version is specified, the debugger displays or deletes only the specified version.

4.4.3 List screen

Purpose: You can use the List screen to display storage areas, session attributes, and debugger symbols at your terminal. The List screen can be used during setup and at runtime.

The specification area of the List screen looks like:

```
LIST: M (M-MEMORY/A-ATTRIBUTES)
MEMORY ONLY:
  BEGIN LIST AT:

      LENGTH.....:          - OR -      END LIST AT:

LIST FORMAT...: B (C-CHARACTER/X-HEX/B-BOTH)
```

Field options: These are the field options for this area:

LIST: M (M-MEMORY/A-ATTRIBUTES)

- M (the default) requests a list of an area of memory specified in the Memory Only section of the screen

- A requests a list of current session attributes; no other options need to be specified on the screen in this case

BEGIN LIST AT:

Specifies the beginning location for the display. The beginning location can include one or more debug expressions resolving to an address for which you have retrieval security.

►► For information on the security methods used by the debugger, see *CA-IDMS Security Administration*. This field is required if a memory display is selected.

LENGTH.....:

Specifies the number of bytes to be displayed.

END LIST AT:

Specifies the ending location for the display. The ending location can include the same debugger entities as those specified for the beginning location.

Important: If a resource is listed and the length or ending location exceeds the resource boundary, the list is truncated at the boundary and the debugger issues a warning message.

When neither Length nor End List At is specified, the length of the display is based on two rules:

- If the debug expression is composed of a single symbol, the data characteristics of the symbol determine the default length. The number of bytes displayed is equal to the length attribute of the symbol.
- If the expression does not have data characteristics, the default length is 16 bytes.

LIST FORMAT..: B (C-CHARACTER/X-HEX/B-BOTH)

- C requests a display in character format
- X requests a display in hexadecimal format
- B (the default) requests a display in both character and hexadecimal format

4.4.4 Resume screen

Purpose: You can use the Resume screen to instruct the runtime system to continue program execution at the next instruction or at another location or to continue standard processing of an abend.

The specification area of the Resume screen looks like:

```
RESUME: E (E-EXECUTION/A-ABEND)
```

```
EXECUTION ONLY:
LOCATION IF OTHER THAN BREAKPOINT:
```

Field options: These are the field options for this area:

RESUME: E (E-EXECUTION/A-ABEND)

Indicates the next action of the runtime system:

- E (the default) requests that execution continue at the next instruction or at another location as indicated by the address specified in the Execution Only section of the screen
- A requests that standard DC/UCF abend processing, including the execution of any STAE exit, should continue

LOCATION IF OTHER THAN BREAKPOINT:

Specifies the location at which execution is to continue. The specified value can be a debug expression that resolves to an address containing a valid instruction or a valid CME (CA-ADS dialogs only).

4.4.5 Set screen

Purpose: You can use the Set screen to modify selected areas of storage and session attributes. The Set screen can be used during setup and at runtime.

The specification area of the Set screen looks like:

```
SET: M (M-MEMORY/C-CHARACTER/X-HEX/B-BOTH)
MEMORY ONLY:
  BEGIN SET MEMORY AT:

  EQUALS.....:

  RESET.....: N (Y-YES/N-NO)
```

Field options: These are the field options for this area:

SET: M (M-MEMORY/C-CHARACTER/X-HEX/B-BOTH)

- **M** (the default) requests modification of the area of memory specified in the Memory Only section of the screen
- The other three options pertain to the setting of session attributes:
 - C requests a display in character format
 - X requests a display in hexadecimal format
 - B requests a display in both character and hexadecimal format

BEGIN SET MEMORY AT:

Specifies the beginning location of the entity to be modified. The beginning location can be a debug expression that resolves to an address for which you have update security. A beginning location value is required when you are updating memory.

If the debug expression is a symbol with data characteristics, the length of the symbol is used in the set. Otherwise, the data characteristics of the source field are used in the set.

Remember: The debugger does not allow a set across resource boundaries.

EQUALS.....:

Indicates the new value that will be assigned to the entity. You can supply an explicit value or a data field name, as in these examples:

```
h'03'
f'9956'
x'f0c4'
c'edit'
p'1234'
'customer-name-0145'
```

The EQUALS field is required when you are updating memory.

RESET.....: N (Y-YES/N-NO)

Indicates the disposition of the original storage value:

- Y requests that the named storage be reset to its original value at the end of the debugging session; this option is not supported for release 10.2 of the debugger
- N (the default) requests that the named storage not be reset to its original value at the end of the debugging session

This option does not affect storage in the debugged program itself since a special copy of the program is loaded for the debugging session.

4.4.6 Snap screen

Purpose: The Snap screen lets you create and write a dump to the DC/UCF log at any point in the debugging session, in order to make a hard copy of storage contents.

Remember: To obtain a hard copy of the Snap dump, use the Batch Command Facility utility.

The specification area of the Snap screen looks like:

```
SNAP: (A-AREA/T-TASK) TITLE:
SKIP: (1-ONE LINE/2-TWO LINES/3-THREE LINES/T-TOP OF NEXT PAGE)
```

```
AREA ONLY:
BEGIN SNAP AT:
```

```
LENGTH: -OR- END SNAP AT:
```

Field options: These are the field options for this area:

SNAP: (A-AREA/T-TASK)

- A requests a dump of the memory area specified in the fields in the Area Only section of the screen
- T requests a dump of all resources associated with the executing task

This is a required field on the Snap screen.

TITLE:

Specifies an optional title for the snap. The title can contain up to 42 characters. Do *not* enclose the title in quotation marks. An apostrophe in the title must be

coded as two single quotes. When a title is not specified, a default title is written to the log:

USER user-id

SKIP: (1-ONE LINE/2-TWO LINES/3-THREE LINES/T-TOP OF NEXT PAGE)

Indicates the carriage control that will be used for placement of the title:

- 1 skips one line
- 2 skips two lines
- 3 skips three lines
- T skips to the top of the next page

If you specify nothing, two lines are skipped.

BEGIN SNAP AT:

Specifies the location at which to begin the snap. The beginning location can be a debug expression that resolves to an address for which you have retrieval security. This field is required when snapping an area.

LENGTH:

Indicates the number of bytes to be snapped.

END SNAP AT:

Indicates the ending location of the snap. The ending location can specify the same types of debug expressions as those used in the Begin Snap At field.

When you do not specify an ending location or a specific length, the default length is based on two rules:

- If the debug expression is composed of a single symbol, the data characteristics of the symbol determine the default length. The number of bytes dumped is equal to the default length of the symbol.
- If the expression does not have data characteristics, the default length is 256 bytes.

4.5 Global help screens

Three available: The debugger provides three global help screens, one each of commands, symbols and control keys.

4.5.1 Usage screen

Top-level screen: The Usage screen is the top-level screen for menu mode. It presents a list of all debugger commands and summarizes the command functions. The Usage screen looks like this:

```

IDMS-DC REL 15.0 ONLINE DEBUGGER *** USAGE ***          SETUP   PAGE 1 OF 4
PROGRAM:          V:          CSECT:
->
          PROCEDURAL COMMANDS.

EXIT.....RETURNS CONTROL TO IDMS-DC/UCF WITHOUT TERMINATING THE CURRENT DEBUGGER
SESSION.
QUIT.....TERMINATES THE DEBUGGER SESSION AND RETURNS CONTROL TO IDMS-DC/UCF.
PROMPT...INVOKES THE PROMPT MODE OF THE DEBUGGER.

          RETRIEVAL COMMANDS.

AT.....ESTABLISHES OR MODIFIES BREAKPOINTS WITHIN A USER PROGRAM.
DEBUG....DESIGNATES, DURING THE SETUP PHASE, THE ENTITY TO BE DEBUGGED OR
INQUIRES ABOUT ENTITIES KNOWN TO THE DEBUGGER.
IOUSER....DISPLAYS THE USER SCREEN THAT IS CURRENT WHEN A BREAKPOINT, PROGRAM
INTERRUPT OR TRAPPED ABEND IS ENCOUNTERED.

NEXT _ ACTIVITY OR _ HELP:
  _ AT      _ LIST      _ SET      _ SNAP      _ RESUME      _ DEBUG      _ WHERE

      _ EXIT      _ PROMPT      _ QUIT      _ IOUSER
HELP _ SCREENS:  _ USAGE      _ SYMBOLS      _ KEYS

```

4.5.2 Symbols screen

Has a specification area: The Symbols screen lets you list program or debugger symbols owned by the entity being debugged. The Symbols screen is the only global help screen with a specification area:

```

      IDMS-DC REL 15.0 ONLINE DEBUGGER *** SYMBOLS ***      SETUP   PAGE 1 OF 1
PROGRAM:          V:          CSECT:
->

SYMBOLS TO DISPLAY: P (P-PROGRAM/D-DEBUGGER)
SYMBOL OR SEARCH STRING:

NEXT _ ACTIVITY OR _ HELP:
  _ AT      _ LIST      _ SET      _ SNAP      _ RESUME      _ DEBUG      _ WHERE

      EXIT      _ PROMPT      _ QUIT      _ IOUSER
HELP SCREENS:  _ USAGE      _ SYMBOLS  _ KEYS

```

Field options: These are the field options for the specification area:

SYMBOLS TO DISPLAY: P (P-PROGRAM/D-DEBUGGER)

Indicates whether program symbols (**P**) or debugger symbols (**D**) for the current entity are to be displayed. The default is **P**. The symbols are listed alphabetically.

SYMBOL OR SEARCH STRING:

Identifies a specific symbol or string that begins the display. When this field does not contain an entry, all specified program or debugger symbols are displayed from the beginning of the list

Example: For example, to begin the display with program symbols prefaced by MIS, you would supply this information on the screen:

```

SYMBOLS TO DISPLAY: p (P-PROGRAM/D-DEBUGGER)
SYMBOL OR SEARCH STRING: mis

```

4.5.3 Keys screen

Installation-specific: The Keys screen provides a list of the current control key assignments for your particular installation. The information displayed on this screen reflects the installation-specific key assignments made with the KEYS statement when the system was generated. The Keys screen contains the most up-to-date information on control key assignments. If an assignment is modified after the system is generated, the Keys screen is also modified automatically.

A sample Keys screen is shown below.

```

      IDMS-DC REL 15.0 ONLINE DEBUGGER *** KEYS ***          SETUP   PAGE 1 OF 1
PROGRAM:          V:          CSECT:
->
  PFKEY ..... ACTIVITY          PFKEY ..... ACTIVITY
  -----
ENTER  PROCESS CURRENT SCREEN    PF5   SYMBOLS SCREEN
CLEAR  PREVIOUS LEVEL            PF6   PFKEYS SCREEN
PA1    REFRESH                   PF7   DISPLAY PREVIOUS PAGE
PA2    EXIT                       PF8   DISPLAY NEXT PAGE
PF1    USAGE SCREEN              PF9   CHANGE TO PROMPT MODE
PF2    UNASSIGNED                PF10  UNASSIGNED
PF3    ACTIVITY SCREEN           PF11  UNASSIGNED
PF4    ACTIVITY HELP SCREEN      PF12  RESERVED

NEXT _ ACTIVITY OR _ HELP:
  _ AT      _ LIST      _ SET      _ SNAP      _ RESUME      _ DEBUG      _ WHERE

      EXIT      _ PROMPT      _ QUIT      _ IOUSER
HELP SCREENS:  _ USAGE      _ SYMBOLS      _ KEYS

```


Chapter 5. Aids for Debugging Assembler, COBOL, and PL/I Programs

5.1 Overview	5-3
5.2 Compiler options	5-4
5.3 COBOL programs	5-5
5.3.1 Preliminary computations	5-5
5.3.2 Sample COBOL online debugger session	5-9
5.4 PL/I programs	5-11
5.4.1 Preliminary computations	5-11
5.4.2 Sample PL/I online debugger session	5-14

5.1 Overview

This chapter discusses online debugger usage with Assembler, COBOL, and PL/I programs. To effectively use the debugger with these languages, specific compiler options must be utilized to produce listings to obtain required information. The compiler options for each programming language are shown in the next topic.

To use the debugger with COBOL or PL/I programs, some preliminary computations must be done to calculate the exact location of variable storage fields or object code to set breakpoints. This chapter contains a discussion of these calculations and sample debugger sessions for both languages.

5.2 Compiler options

The following table shows the compiler options which provide the information required to use the online debugger to analyze your program.

Language	Object Code	Variable Storage
VS-COBOL	PMap or CLIST	DMap
VS-COBOL II COBOL for OS/390 and VM/ESA COBOL for VSE/ESA	LIST or OFFSET	MAP
PL/I	LIST, XREF, and OFFSET	STORAGE and MAP
Assembler	LIST	LIST

5.3 COBOL programs

This section discusses the preparation that is necessary before beginning to debug a COBOL program and provides a sample COBOL debugging session.

Note: The discussion and sample debugger session that follow are for a program compiled under the VS-COBOL compiler. The basic principals are the same for other compiler levels such as VS-COBOL II or COBOL for OS/390 and VM/ESA. Some specific differences are noted. For more information on register conventions and program structure, refer to the appropriate IBM documentation.

5.3.1 Preliminary computations

Before beginning the debugging process, it is recommended to determine the breakpoints that you want to set and the storage locations that you want to examine.

Breakpoints: To determine the hexadecimal offset of an executable program instruction at which you want to set a breakpoint, perform the following steps:

1. Examine the COBOL compiler portion of your listing and record the line number of the statement at which you want to set the breakpoint:

```

00787      *
00788      *   OBTAIN EMPLOYEE DB-KEY IS EMP-DBKEY
00789      *           ON ANY-STATUS
00790      *           MOVE 0 TO DCNUM1 DCNUM2 DCFLG1 DCFLG2
00791      *           MOVE 0028 TO DML-SEQUENCE
00792      *           CALL 'IDMSCOBI' USING SUBSCHEMA-CTRL
00793      *                   IDBMSCOM (06)
00794      *                   SR415
00795      *                   EMP-DBKEY
00796      *                   IDBMSCOM (43)
00797      *           IF NOT ANY-STATUS PERFORM IDMS-STATUS;
00798      *           ELSE
00799      *                   NEXT SENTENCE.
00800      *           IF DB-REC-NOT-FOUND
00801      *                   MAP OUT USING DCTEST01
00802      *                   MESSAGE IS EMP-NOT-FOUND-MESS
00803      *                   TO EMP-NOT-FOUND-MESS-END
00804      *                   DETAIL CURRENT

```

2. Examine the condensed listing (CLIST) portion of the COBOL compiler listing, locate the previously recorded COBOL line number, and record its corresponding hexadecimal displacement value:

```

CONDENSED LISTING
.
.
785  MOVE      001CCC      786  GO      001CD0
790  MOVE      001CD6      791  MOVE    001CEE
792  CALL      001CF4      797  IF      001D3E
797  PERFORM   001D4C      800  IF      001D74
805  MOVE      001D80      806  MOVE    001D98
807  MOVE      001D9E      808  MOVE    001DA4
.
.

```

WORKING-STORAGE SECTION variables: To determine the register assignment and offset of WORKING-STORAGE SECTION variables, perform the following steps:

1. Locate the register assignment portion of the COBOL compiler listing and record the base locator (BL) number that corresponds to each register listed:

```
REGISTER ASSIGNMENT
REG 6 BL =1
```

Note: For some WORKING-STORAGE or LINKAGE SECTION fields, there may not be a fixed register which always points to the base locator for linkage (BLL) cell. However, the BL cell is at a given offset from the beginning of the TGT.

For compiler levels prior to COBOL for OS/390 and VM/ESA, register 13 usually points to the TGT at runtime. For COBOL for OS/390 and VM/ESA and higher-level compilers, register 9 usually points to the TGT at runtime.

A copy of the TGT and WORKING STORAGE is allocated in the CA-IDMS storage pools for each task at runtime. Therefore, you must not use the TGT or WORKING STORAGE in the program pool.

2. Locate the data map (DMAP) portion of the COBOL compiler listing and record the displacement value and register assignment for each variable that you want to examine during the debugging process:

DNM=1-364	01	LONGTERM-TEST	BL=1	038	DNM=1-364	DS 4C	COMP
DNM=1-387	01	EMP-DBKEY	BL=1	040	DNM=1-387	DS 1F	COMP
DNM=1-406	01	FIRST-PAGE-SW	BL=1	048	DNM=1-406	DS 1C	DISP
DNM=1-432	88	LESS-THAN-A-PAGE			DNM=1-432		
		.					
		.					
DNM=4-276	01	SUBSCHEMA-CTRL	BL=1	260	DNM=4-276	DS 0CL320	GROUP
DNM=4-303	02	PROGRAM-NAME	BL=1	260	DNM=4-303	DS 8C	DISP
DNM=4-325	02	ERROR-STATUS	BL=1	268	DNM=4-325	DS 4C	DISP
DNM=4-350	88	DB-STATUS-OK			DNM=4-350		
DNM=4-376	88	ANY-STATUS			DNM=4-376		
DNM=4-399	88	ANY-ERROR-STATUS			DNM=4-399		
DNM=4-425	88	DB-END-OF-SET			DNM=4-425		
DNM=4-452	88	DB-REC-NOT-FOUND			DNM=4-452		
DNM=6-028	02	DBKEY	BL=1	26C	DNM=6-028	DS 4C	COMP
DNM=6-043	02	RECORD-NAME	BL=1	270	DNM=6-043	DS 16C	DISP
		.					

LINKAGE SECTION variables: To determine the location of LINKAGE SECTION variables, perform the following steps:

1. Examine the memory map portion of the COBOL compiler listing and locate the hexadecimal displacement values for the TGT and for the base locator for linkage (BLL) cells:

MEMORY MAP		
TGT		00868
SAVE AREA		00868
SWITCH		008B0
TALLY		008B4
SORT SAVE		008B8
ENTRY-SAVE		008BC
.		
.		
TEMP STORAGE-3		00A78
TEMP STORAGE-4		00A78
BLL CELLS		00A78
VLC CELLS		00A8C
.		
.		

- Perform the following calculation to determine the displacement value for the BLL cells:

$$\text{BLL CELLS} - \text{TGT} = \text{displacement for BLL cells}$$

$$\text{X'A78'} - \text{X'868'} = \text{X'208'}$$

- Locate the BLL number for the first LINKAGE SECTION variable from the DMAP portion of the compiler listing:

DNM=14-361	01	PASS-DEPT-INFO	BLL=3	000	DNM=14-361	DS OCL5	GROUP
DNM=14-391	02	PASS-DEPT-ID	BLL=3	000	DNM=14-391	DS 4C	DISP-NM
DNM=14-416	02	PASS-DEPT-INFO-END	BLL=3	004	DNM=14-416	DS 1C	DISP
DNM=14-444	01	ERROR-DATA	BLL=4	000	DNM=14-444	DS OCL9	GROUP
DNM=14-467	02	ERROR-DEPT-ID	BLL=4	000	DNM=14-467	DS 4C	DISP-NM
DNM=15-000	02	ERROR-MESSAGE-CODE	BLL=4	004	DNM=15-000	DS 4C	DISP
DNM=15-031	02	ERROR-DATA-END	BLL=4	008	DNM=15-031	DS 1C	DISP

- Save the displacement values of the BLL cells and the BLL numbers of LINKAGE SECTION variables for use during the runtime phase to obtain the absolute address for LINKAGE SECTION values.

You can use the following table to record displacement information before starting a debugger session.

Program Name _____		Comment _____	
Line Number	Displacement		
Variable-Storage Field	Displacement	Base Register	
Linkage Section Field	BLL Displacement	Absolute Address	

5.3.2 Sample COBOL online debugger session

To use the online debugger with a DC/UCF VS-COBOL program, perform the steps shown below. The steps may need to be modified for other compilers such as VS-COBOL II or COBOL for OS/390; however, the basic methodology is the same.

1. Compile the program with the DMAP and CLIST compiler options before defining it to the DC/UCF system.

Note: To obtain the complete Assembler source code, substitute CLIST with PMAP.

2. Record breakpoint and storage displacements, as explained earlier under COBOL Programs.

3. Initiate the debugger session by entering the DEBUG task code from the DC/UCF system. The DEBUG> prompt displays indicating that the debugger is in control:

```
ENTER NEXT TASK CODE:
debug
```

```
DEBUG>
```

4. Specify the program to be debugged by entering DEBUG followed by the program name. The debugger verifies the program name:

```
DEBUG>
debug testprog
```

```
DEBUG TESTPROG
DEBUG> DEBUGGING INITIATED FOR TESTPROG VERSION 1
DEBUG>
```

5. Establish breakpoints by issuing the AT command followed by a dollar sign, which signifies the address of the beginning of the program; follow the dollar sign with the command's hexadecimal offset. The debugger verifies the establishment of the breakpoint:

```
DEBUG>
at $ + @1d4c
```

```
AT @1D4C
AT> @1D4C ADDED
DEBUG>
```

After all breakpoints have been set, leave the setup phase of the debugger session by issuing the EXIT command:

```
DEBUG>
exit
```

6. Initiate the runtime phase by issuing the task code that invokes the task in which the program participates:

```
ENTER NEXT TASK CODE:
deptmod
```

When a breakpoint is encountered at runtime, the debugger assumes control and identifies the address, program, and the debugger expression that was used to establish the breakpoint:

```
AT OFFSET @1D4C IN TESTPROG EXPRESSION @1D4C
DEBUG>
```

7. Examine program variable storage by issuing LIST commands. Use indirect addressing and the previously noted register and offset:

```
list %:r6 + @268 32
```

```
LIST %:R6 + @268 32
00140270 E3C5E2E3 D7D9D6C7 F0F0F0F0 3D3D4F06 *TESTPROG0000..|. *
00140280 C4C5D7C1 D9E3D4C5 D5E34040 40404040 *DEPARTMENT *
```

To examine LINKAGE SECTION variables, perform the following steps:

- a. List the contents of register 13 plus the previously determined BLL displacement using indirect addressing:

```
DEBUG>
list %:rR13 + @208

LIST %:R13 + @208
          (BLL1)  (BLL2)  (BLL3)  (BLL4)
001499E0 00000000 00000000 00000000 00149AC8 *.....H*
```

Each BLL is 4 bytes long. Note the absolute address located in the BLL for the field that you want to display.

- b. List the absolute address to display the LINKAGE SECTION values:

```
DEBUG>
LIST @149ac8 16
00149AC8 F1F1F1F1 C4C5D7E3 00000000 00000000 *1111DEPT.....*
```

8. Enter the RESUME command from the DEBUG> prompt to continue program execution:

```
DEBUG>
resume
```

9. Enter the QUIT command from the DEBUG> prompt to end a debugger session:

```
DEBUG>
quit

QUIT
QUIT DEBUGGER
ENTER NEXT TASK CODE:
```

5.4 PL/I programs

This section discusses the preparation that is necessary before beginning to debug a PL/I program and provides a sample PL/I debugging session.

Note: The discussion and sample debugger session that follow are for a program compiled under the PL/I Version 2.3 compiler. The basic principals are the same for other compiler levels such as PL/I for OS/390 and VM/ESA. For more information on register conventions and program structure, refer to the appropriate IBM documentation.

5.4.1 Preliminary computations

Before beginning the debugging process, it is recommended to determine the break-points that you want to set and the storage locations that you want to examine.

Breakpoints: To determine the hexadecimal offset of an executable program instruction at which you want to set a breakpoint, perform the following steps:

1. Examine the cross-reference table portion of your link-edit listing for an entry in the form program-name1. Record the hexadecimal offset listed under ORIGIN:

CROSS REFERENCE TABLE					
CONTROL SECTION	NAME	ORIGIN	LENGTH	ENTRY NAME	LOCATION
	PLISTART	00	50		
	PLIMAIN	50	8	PLICALLA	6
	*PLIPROG2	58	394		
	*PLIPROG1	3F0	EB4		
	IDMSPLI	12A8	284	PLI3PROG	3F8

2. Examine the PL/I compiler portion of your listing and record the line number of the statement at which you want to set the breakpoint:

```

133      WORK_LAST = EMP_LAST_NAME_0415;
134      WORK_FIRST = EMP_FIRST_NAME_0415;
                                     /*
MAP OUT (DCTEST01) OUTPUT DATA YES
MESSAGE (INITIAL_INSTRUCTIONS_MSG_1)
LENGTH (25)
DETAIL NEW KEY (DBKEY).
                                     */
135      /* IDMS PLI/I DML EXPANSION */ DO;
136      DML_SEQUENCE=0013;
137      DCCFLG1=0;
138      DCCFLG1=13;
139      DCCFLG2=16;
140      DCCFLG3=0;
141      DCCFLG4=4;
142      DCCFLG5=72;
143      DCCFLG6=0;

```

3. Examine the Assembler listing generated by the LIST option, locate the previously recorded PL/I line number, and record its corresponding hexadecimal displacement value:

```

* STATEMENT NUMBER 136
0006AA 41 80 7 21C      LA  8,SUBSCHEMA_CTRL.D
                        CCALIGN_AREA.FILLE
                        R0001
0006AE 58 40 3 124      L   4,292(0,3)
0006B2 50 40 8 008      ST  4,SSC_ERRSAVE_AREA
                        .DML_SEQUENCE
    
```

4. Add the origin offset and the breakpoint instruction's hexadecimal displacement to obtain the breakpoint address:

$$X'3F0' + X'6AA' = X'A9A'$$

AUTOMATIC variables: To determine the offset of AUTOMATIC variables, locate the variable storage map and record the displacement value for each variable that you want to examine during the debugging process:

MAP_WORK_REC	1	796	31C	AUTO
WORK_DEPT_ID	1	796	31C	AUTO
WORK_EMP_ID	1	800	320	AUTO
WORK_FIRST	1	804	324	AUTO
WORK_LAST	1	814	32E	AUTO
WORK_ADDRESS	1	829	33D	AUTO
WORK_STREET	1	829	33D	AUTO
WORK_CITY	1	849	351	AUTO
WORK_STATE	1	864	360	AUTO
WORK_ZIP	1	866	362	AUTO
WORK_DEPT_NAME	1	871	367	AUTO

You can locate AUTOMATIC variables at runtime through register 13.

STATIC INTERNAL variables: To determine the location of STATIC INTERNAL variables, examine the static internal storage map to find the hexadecimal offset for each variable that you want to examine during the debugging process.

You can locate STATIC INTERNAL variables at runtime through register 3.

You can use the following table to record displacement information before starting a debugger session.

5.4.2 Sample PL/I online debugger session

To use the online debugger with a DC/UCF PL/I program, perform the following steps:

1. Compile the program with the LIST, OFFSET, XREF STORAGE, and MAP compiler options before defining it to the DC/UCF system.
2. Record breakpoint and storage displacements, as explained above.
3. Initiate the debugger session by entering the DEBUG task code from the DC/UCF system. The DEBUG> prompt displays indicating that the debugger is in control:

```
ENTER NEXT TASK CODE:  
debug
```

```
DEBUG>
```

4. Specify the program to be debugged by entering DEBUG followed by the program name. The debugger verifies the program name:

```
DEBUG>  
debug plipro
```

```
DEBUG PLIPROG  
DEBUG> DEBUGGING INITIATED FOR PLIPROG VERSION 1  
DEBUG>
```

5. Establish breakpoints by issuing the AT command followed by a dollar sign, which signifies the address of the beginning of the program; follow the dollar sign with the command's hexadecimal offset. The debugger verifies the establishment of the breakpoint:

```
DEBUG>  
at $ + @a9a
```

```
AT @A9A  
AT> @A9A ADDED  
DEBUG>
```

After all breakpoints have been set, leave the setup phase of the debugger session by issuing the EXIT command:

```
DEBUG>  
exit
```

6. Initiate the runtime phase by issuing the task code that invokes the task in which the program participates:

```
ENTER NEXT TASK CODE:  
deptmod
```

When a breakpoint is encountered at runtime, the debugger assumes control and identifies the address, program, and the debugger expression that was used to establish the breakpoint:

```
AT OFFSET @A9A IN PLIPROG EXPRESSION @BDE  
DEBUG>
```

7. Examine program variable storage by issuing LIST commands. Use indirect addressing and the previously noted register and offset:

```
list %:r13 + @31c 32
```

```
LIST %:R13 + @31C 32
```

```
001DB7F4 F3F2F0F0 F0F0F0F4 C8C5D9C2 C5D9E340 *32000004HERBERT*
001DB804 4040C3D9 C1D5C540 40404040 40404040 * CRANE *
```

If your program contains any nested procedures or begin blocks, you will need to navigate the chain of dynamic storage areas (DSAs) to obtain the correct variable-storage base address. To navigate the DSA chain for nested procedures or begin blocks, list the contents of register 13 to determine the DSA for the current level of nesting:

```
list %:r13
```

```
LIST %:R13
```

```
001C7A30 84200000 001C7948 00000000 5E422A20 *D.....;...*
```

For subsequent levels of nesting, perform the following steps:

- a. List the absolute address which is located 4 bytes off of the previously displayed line:

```
list @1c7948
```

```
LIST @1C7948
```

```
001C7948 84200000 001C74D8 00000000 4E4227EC *D.....Q....+...*
```

- b. List AUTOMATIC variable-storage values after the final level of nesting has been reached. Use the absolute address as the base address, which is located 4 bytes off of the display:

```
DEBUG>
```

```
list 1c74d8 + @31c 32
```

```
LIST 1C74D8 + @31C 32
```

```
001C77F4 F3F2F0F0 F0F0F0F4 C8C5D9C2 C5D9E340 *32000004HERBERT *
001C7804 4040C3D9 C1D5C540 40404040 40404040 * CRANE *
```

To examine variables defined as BASED storage, perform the following steps:

- a. List the contents of the associated pointer variable using indirect addressing:

```
DEBUG>
```

```
list %:r13 + @d4
```

```
LIST %:R13 + @D4
```

```
001499E0 00149AC8 00000000 00000000 00000000 *...H.....*
```

- b. List the absolute address to display the BASED variable's values:

```
DEBUG>
```

```
LIST @149ac8 16
```

```
00149AC8 F1F1F1F1 C4C5D7E3 00000000 00000000 *1111DEPT.....*
```

8. Enter the RESUME command from the DEBUG> prompt to continue program execution:

```
DEBUG>  
resume
```

9. Enter the QUIT command from the DEBUG> prompt to end a debugger session:

```
DEBUG>  
quit
```

```
QUIT  
QUIT DEBUGGER  
ENTER NEXT TASK CODE:
```

Index

Special Characters

#ABEND 1-11

A

absolute address notation 2-5

address symbols

at sign (@) 2-5

cent sign (¢) 2-6

dollar sign (\$) 2-5

Assembler programs

compiler options 5-4

debugging aids 5-3

LIST option 5-11

AT command examples 3-5

AUTOMATIC variables 5-12

B

base locator for linkage cell 5-6

base locator number 5-6

BASED storage variables 5-15

BL number 5-6

BLL cell 5-6

breakpoint counter 3-5

breakpoints

bypassing 3-5

encountering 1-11, 1-12, 3-5

listing 3-5

modifying 3-5

removing 3-5

setting 1-4, 1-11, 3-5, 5-5, 5-11

status 3-5

using 3-4

C

CLIST 5-4

COBOL programs

compiler options 5-4

debugging aids 5-3

sample debugger session 5-9

command format 2-18

commands

AT 3-4, 5-9, 5-14

DEBUG 3-7, 5-9, 5-14

EXIT 3-9, 5-9, 5-14

formatting 2-18

commands (*continued*)

IOUSER 3-10

LIST 3-11, 5-10, 5-15

MENU 3-14

modifying 2-15

PROMPT 3-15

QUALIFY 3-16

QUIT 3-18, 5-10, 5-16

RESUME 3-19, 5-10, 5-15

SET 3-20

SNAP 3-24

WHERE 3-26

compiler options 5-4

control block 3-18

currency

dialog process 3-16

inquire about 3-16

load module 3-7

process 2-8

program 1-14

reset 3-16

current process address 2-6

current screen 1-13

D

data characteristics 3-12

expressions with 2-11

expressions without 2-12

table 2-11

data fields

displaying 2-7

qualifying 2-7

data values

numeric 2-17

strings 2-17

DEBUG

command 1-7

prompt 1-6, 1-8

task code 1-6, 1-8

debug expressions

data characteristics 3-12

default length 2-11

debug expressions, components of

address symbols 2-5

debugger symbols 2-5

general registers 2-4

program status word (PSW) 2-4

debug expressions, components of (*continued*)

program symbols 2-7

special operators 2-9

standard operators 2-9

system symbols 2-4

user symbols 2-6

debugger commands 3-3—3-26

formatting 2-18

modifying 2-15

parsing 2-14

debugger control blocks 1-7

debugger features 1-4

breakpoints 1-4

debugger implementation 1-6—1-8, 5-3

debugger labels

cent sign (¢) 2-6

dollar sign (\$) 2-5

debugger markers

at sign (@) 2-5

debugger session

Assembler programs 5-3

COBOL program, sample 5-9

COBOL programs 5-3

compiler options 5-3

definition 1-6

initiating 1-6, 3-8, 5-3

leaving 1-7, 3-18

length considerations 1-13

menu mode 1-8

PL/I program, sample 5-14

PL/I programs 5-3

prompt mode 1-6

runtime phase 1-11

setup phase 1-11

terminating 1-7, 3-9

debugger symbols 2-4

debugger variables

displaying 3-11

defining entities

to DC/UCF 1-6

to the debugger 1-6

display length 2-12

DMAP 5-4

E

efficient use of resources 1-13

expression components 2-4—2-10

G

general registers 2-4

L

length attributes 2-11

line numbers 2-8

LINKAGE SECTION variables 5-6

LIST 5-4

load address 2-5

location inquiry 3-26

M

MAP 5-4

memory

See also storage values

displaying 1-4

modifying 1-4

menu mode 1-8

multiple expressions 2-4

N

numeric values

decimal 2-17

fullword 2-17

halfword 2-17

hexadecimal 2-17

O

OFFSET 5-4

P

parsing rules 2-14

performance standards 1-13

PL/I programs

compiler options 5-4

debugging aids 5-3

sample debugger session 5-14

PMAP 5-4

process currency 2-8

processing overhead 1-13

program checks 1-11

program currency 1-14, 3-7

program status word (PSW) 2-4

program symbols

data field names 2-7

line numbers 2-8

program symbols (*continued*)
 qualifying 2-8
prompt mode 1-6

R

resource
 boundary 3-11
 display truncation 3-11
rollbacks 3-9, 3-18
runtime phase 1-11
 commands 1-12

S

session attributes
 displaying 3-12, 3-22
 setting 3-22
session modes
 menu 1-8, 3-14
 prompt 1-6, 3-15
setup phase 1-11
snap dumps 1-4, 3-24
special characters
 at sign (@) 2-5
 cent sign (¢) 2-6
 dollar sign (\$) 2-5
 percent sign (%) 2-9
special operators 2-9
standard operators 2-9
STATIC INTERNAL variables 5-12
STORAGE 5-4
storage values
 displaying 3-11
 modifying 3-20
string values
 character 2-17
 hexadecimal 2-17
 numeric 2-17
symbolic line numbers 2-8
system symbols 2-4

U

user symbols 2-6

V

valid addresses 1-14
 See also command elements
valid breakpoints
 See breakpoints

verification messages 1-7, 1-12, 1-14

W

WORKING-STORAGE SECTION variables 5-6

X

XREF 5-4

