AEG

User's Guide GLS™ Symbolic Debugger (mdb™)

216–856007–001

MODCOMP

AEG

User's Guide

GLS™ Symbolic Debugger (mdb™)

MODCOMP

Manual History

Manual Order Number: 216–856007–001

Title: User's Guide, GLS Symbolic Debugger (mdb)

Revision Level	Date Issued	Description
000	11/89	Initial Issue.
001	05/90	Change. Corrected pathnames of installed directories to reflect INSTALL script.

Contents subject to change without notice.

Preface

Audience

This manual is written for experienced system programmers who are using the General Language System (GLS™) compilers. Users should be familiar with AT&T UNIX® System V or REAL/IX™ Operating System terminology.

Subject

This manual tells how to install and use the GLS Symbolic Debugger (**mdb**™). The **mdb** debug windows incorporate the full functionality of AT&T's UNIX **sdb(1)** command options. A section describing **mdb** rules and syntax is also included.

Product Requirements

Use the GLS Symbolic Debugger with programs processed by one of the GLS compilers. **mdb** runs under the REAL/IX Operating System on the CLASSIC® Tri-Dimensional™ Model 97xx computer system.

Related Publications

Refer to the following manuals for additional information. When you order additional manuals, use the manual order number listed below. The most current revision level (REV) will be shipped.

Manual Order Number	Manual Title
215-856001-*REV*	GLS FORTRAN Interface to System Services Library Reference Manual
210-856001-*REV*	GLS FORTRAN Language Reference Manual
216-856005-*REV*	GLS Programming Guide for 97xx Systems
216-856007-*REV*	GLS Symbolic Debugger User's Guide
205-855001-*REV*	REAL/IX Operating System, 97xx Systems Concepts and Characteristics
211-855001-*REV*	REAL/IX Operating System, 97xx Systems Commands and Utilities Reference Manual
211-854002-*REV*	REAL/IX Operating System, 97xx Systems System Calls, Library Routines, and Files Reference Manual

MODCOMP Service and Assistance

MODCOMP® offers a variety of programs and services that demonstrate our commitment to customer satisfaction. Our Technical Education department provides comprehensive hands–on instruction either at our facilities or at customer–designated sites. Our worldwide field service organization is ready to provide installation assistance, free service during the warranty period, and flexible service programs tailored to your requirements.

Questions, Problems, and Suggestions

Your MODCOMP sales and service representatives can help you with any questions, problems, or suggestions you may have regarding our products and services. In addition, for your convenience MODCOMP maintains toll–free telephone numbers at which we can be reached for questions, problems, and suggestions. Please feel free to use the following numbers:

- ❑ **For questions, sales information, or suggestions**: in the U.S. and Canada, 1–800–255–2066 (In countries outside the U.S. and Canada, please call your regional sales support office or 1–305–974–1380 extension 1800 worldwide.)

- ❑ **For service**: in Florida, 1–800–432–1405; in the U.S., 1–800–327–8928; in Canada, 1–416–890–0666 (In countries outside the U.S. and Canada, please call your regional service/support office.)

- ❑ **For Technical Education information**: in the U.S., 1–305–977–1708 (In countries outside the U.S., please call your regional support office.)

For comments about documentation, please use the response form at the back of this manual.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The GLS™ Symbolic Debugger, **mdb™**, uses a special set of symbols[1] to establish a connection between source code and executable instructions. The compile and link process typically destroys the ability to trace back to the source level. **mdb** restores this connection by providing a high level debugging environment with a window interface that allows you to pause execution at breakpoints, single–step through the program, and examine register or memory contents.

mdb is powerful, but easy to access. As few as five commands provide the user with debugging capabilities. Once you are familiar with the basic debugger functions, the **mdb** user interface and command set make it possible to solve program logic problems with minimum effort. Refer to the section "Using mdb" for the **mdb** command descriptions.

The GLS Symbolic Debugger includes the following features:

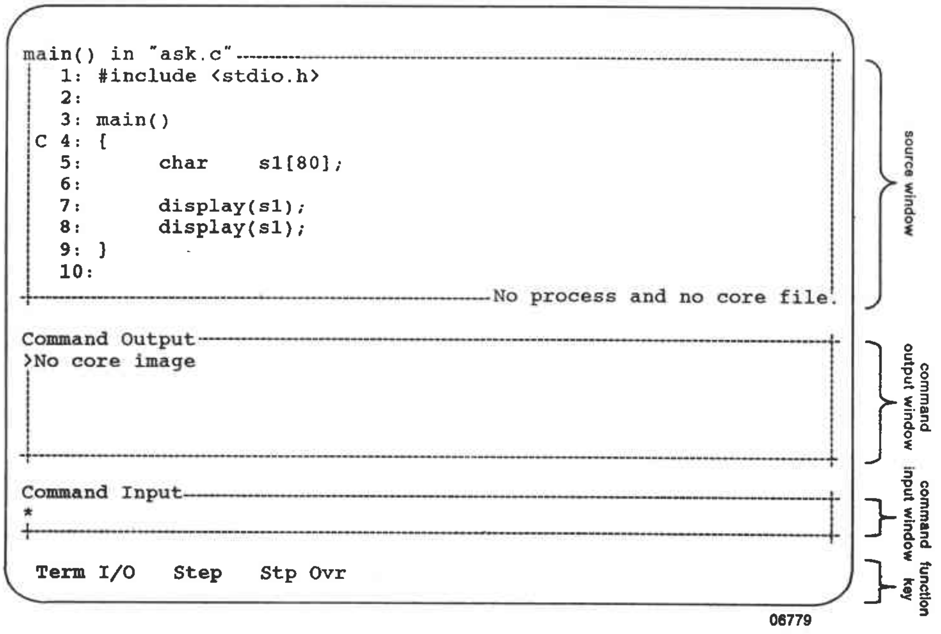
- A user–friendly window interface
- On–line help facility
- Log file capability
- Command files for commonly used **mdb** command sequences.

[1] Symbols are saved during each stage of the compile, assemble, and link translation process.

The Window–Oriented Screen Interface

Based on the standard UNIX® library **curses**(3X) package, the window interface is terminal independent. This means you can use any intelligent terminal if you have an entry in the **terminfo**(4) directory for that terminal.

When you execute **mdb**, you are presented with the initial *Standard Screen*. This screen includes the *Source, Command Output,* and *Command Input* subwindows.



```
main() in "ask.c"-------------------------------------------------------+
|  1: #include <stdio.h>                                                |
|  2:                                                                   |
|  3: main()                                                            |
|C 4: {                                                                 |
|  5:      char    s1[80];                                              |
|  6:                                                                   |
|  7:      display(s1);                                                 |
|  8:      display(s1);                                                 |
|  9: }                                                                 |
| 10:                                                                   |
+---------------------------------------------No process and no core file.

Command Output----------------------------------------------------------+
>No core image                                                          |
|                                                                       |
|                                                                       |
+-----------------------------------------------------------------------+

Command Input-----------------------------------------------------------+
*                                                                       |
+-----------------------------------------------------------------------+

 Term I/O    Step    Stp Ovr
```

Figure 1–1. Standard Screen

The Source Window displays the current source procedure and file name, active breakpoints in the current procedure, the current line, last line executed, and the top of the stack trace.

The Command Output window displays debug process output. The output window implements a subset of the REAL/IX™ pg(1) command options in the scrolling function. Note that the message `>No core image` in the Command Output Window means the user did not request a core image file when executing **mdb**. Refer to the section "Calling mdb" in Chapter 4 .

The Command Input window accepts all **mdb** command line input and supports a history function that allows you to edit and reissue commands.

Window Interface Commands

The GLS Symbolic Debugger package provides full symbolic debugging capabilities, as well as menu–driven screens and window–oriented commands that make **mdb** easy to use.

Control Keys

In this document, a bold character preceded by an uparrow symbol (^) means press the control key while striking the specified character key. The following control keys have been added to provide enhanced functionality:

- ^**P** dumps the current screen to the file *./sdump.dmb*.
- ^**R** clears and redraws the screen.
- ^**E** calls up the on–line help window. Refer to the section "The Window Cycle" for a discussion of the on–line help windows.

Function Keys

Function keys are provided to eliminate keystrokes for the following commands (refer to Figure 1–1). You *must* match the *terminfo* TERM environment variable with your terminal type.

- The `Term I/O` function corresponds to the **<F1>** key – this opens the Terminal I/O Window.
- The `Step` function corresponds to the **<F2>** key – this executes the **s** (step) command.
- The `Stp Ovr` function corresponds to the **<F3>** key – this executes the **S** (stepover) command.

The Window Cycle

mdb supports the ability to cycle from one window to the next; each window includes a pop–up window that provides on–line help.

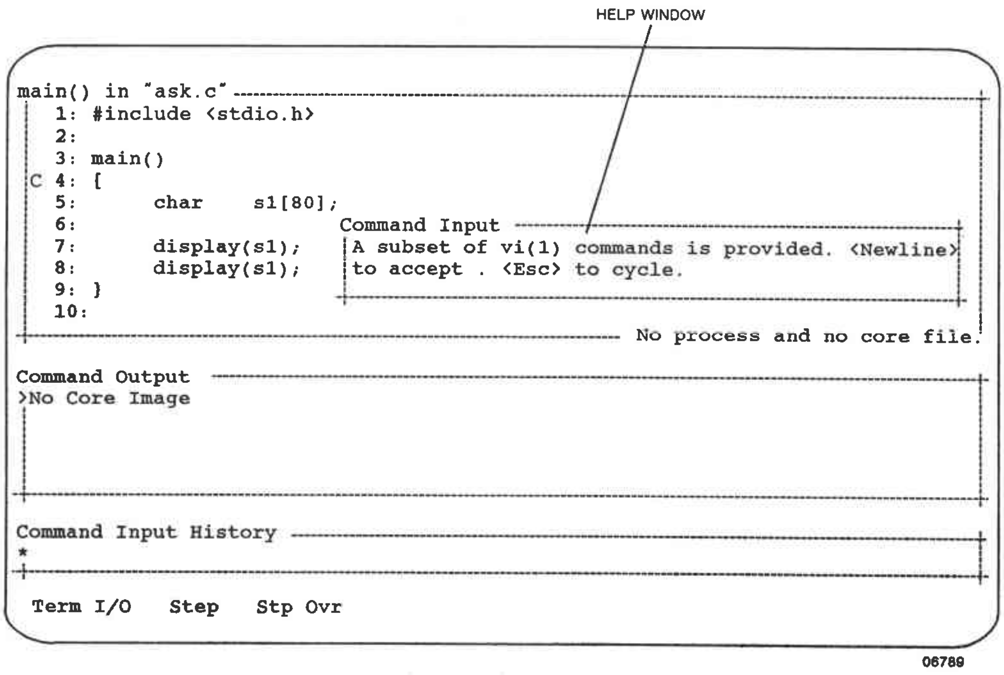
To access each screen press the escape key (**<ESC>**). Press ^**E** to activate the help window for the currently active window. Press the carriage return key (**<CR>**)[1] to exit a window; this returns you to command input mode. Each time you press **<ESC>**, you cycle to the next screen.

Cycle Position	Active Window
1	Command Input
2	Command Output
3	System
4	Terminal I/O

[1] The carriage return key is also known as the **<ENTER>** or **<NEWLINE>** key.

On-line help is available through the help windows. The help windows provide the user with information and/or instructions that pertain to the currently active window. To access on-line help for each window press ^E. The following screens illustrate the four **mdb** windows and their associated help windows.

- The *history* function allows the user to reissue and edit previous commands to the Command Input Window. Commands are edited using a subset of the **vi**(1) editor.



```
main() in "ask.c" -----------------------------------------------------------
   1: #include <stdio.h>
   2:
   3: main()
C  4: {
   5:      char    s1[80];
   6:                         Command Input ------------------------------
   7:      display(s1);      |A subset of vi(1) commands is provided. <Newline>
   8:      display(s1);      |to accept . <Esc> to cycle.
   9: }
  10:
----------------------------------------------------- No process and no core file.

Command Output ------------------------------------------------------------
>No Core Image

Command Input History ----------------------------------------------------
*
```

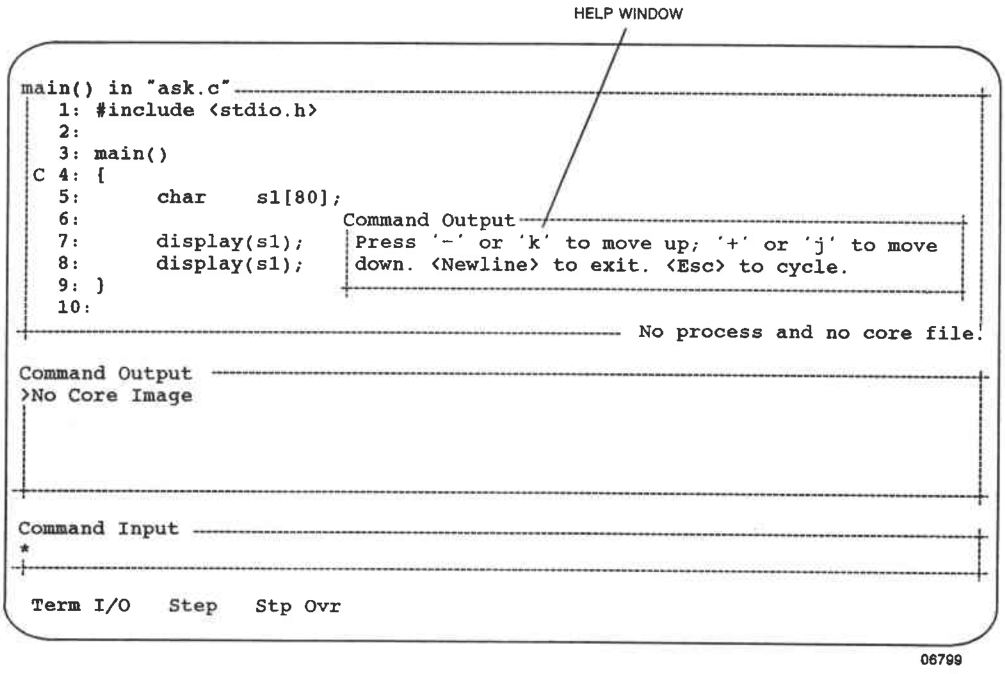
Figure 1-2. Active Command Input Window with History Help Window

Refer to Table 1-1 for the **vi**(1) editor commands that are supported in the Command Input Window.

Table 1–1. Supported vi Commands

vi commands supported by command input	
h, j, k, l	move left, down, up, right,
x	delete a character
ESC	end insert mode
+	move cursor to next line
–	move cursor to previous line
0	move cursor to beginning of line
$	move cursor to end of line
^H	backspace
spacebar	advance one space
a	add after cursor
A	append to end of line
i	Insert before cursor
r*x*	replace character with *x*

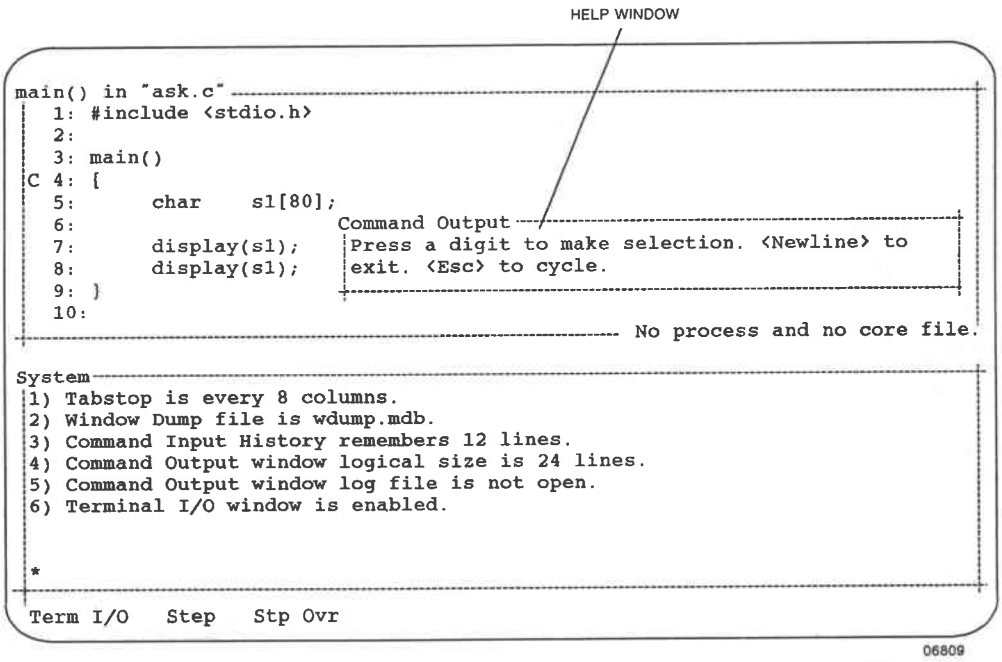
- The *scrolling* function allows the user to review debugger output information by moving forward and back in the Command Output window.



```
main() in "ask.c"
   1: #include <stdio.h>
   2:
   3: main()
C  4: {
   5:      char    s1[80];
   6:                               Command Output
   7:      display(s1);             Press '-' or 'k' to move up; '+' or 'j' to move
   8:      display(s1);             down. <Newline> to exit. <Esc> to cycle.
   9: }
  10:
                                                   No process and no core file.

Command Output
>No Core Image

Command Input
*

 Term I/O   Step   Stp Ovr
```

Figure 1–3. Active Command Output Window with Scrolling Function

- The *system default* function accesses the System Window. The System Window allows the user to modify **mdb** system parameters.



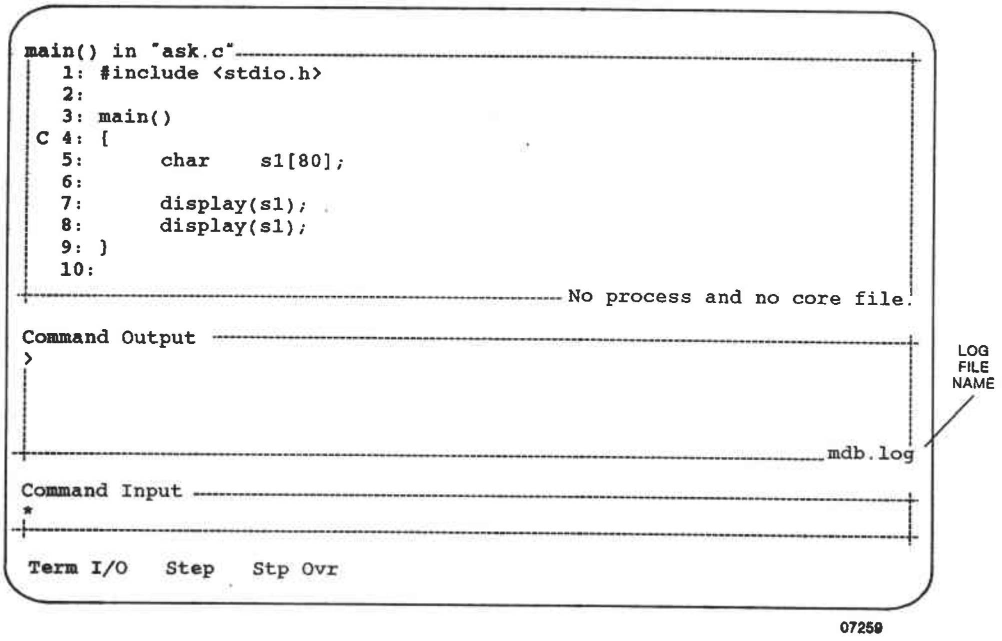
```
main() in "ask.c"
  1: #include <stdio.h>
  2:
  3: main()
C 4: {
  5:       char    s1[80];
  6:                        Command Output
  7:       display(s1);     Press a digit to make selection. <Newline> to
  8:       display(s1);     exit. <Esc> to cycle.
  9: }
 10:
                                                 No process and no core file.
System
1) Tabstop is every 8 columns.
2) Window Dump file is wdump.mdb.
3) Command Input History remembers 12 lines.
4) Command Output window logical size is 24 lines.
5) Command Output window log file is not open.
6) Terminal I/O window is enabled.


*
 Term I/O   Step   Stp Ovr
```

Figure 1–4. Active System Window with Help Window

When you make a selection from the System Window menu, **mdb** provides further instructions and/or waits for the related input. Refer to following sections for additional information about menu items **5** and **6**.

Menu Item 5: Log Files
The contents of the Command Output Window are recorded in the log file. The log file is opened and closed in the System Parameters Window (refer to Figure 1–4). If a log file exists, the name of the log file appears in the lower right–hand corner of the Command Output Window (refer to Figure 1–5).

If you select item number **5** from the System Window menu, you are prompted to enter the pathname. ***file*.log** is like any other REAL/IX™ file. After you create the file, it resides either in the current working directory or in the directory specified by the pathname.



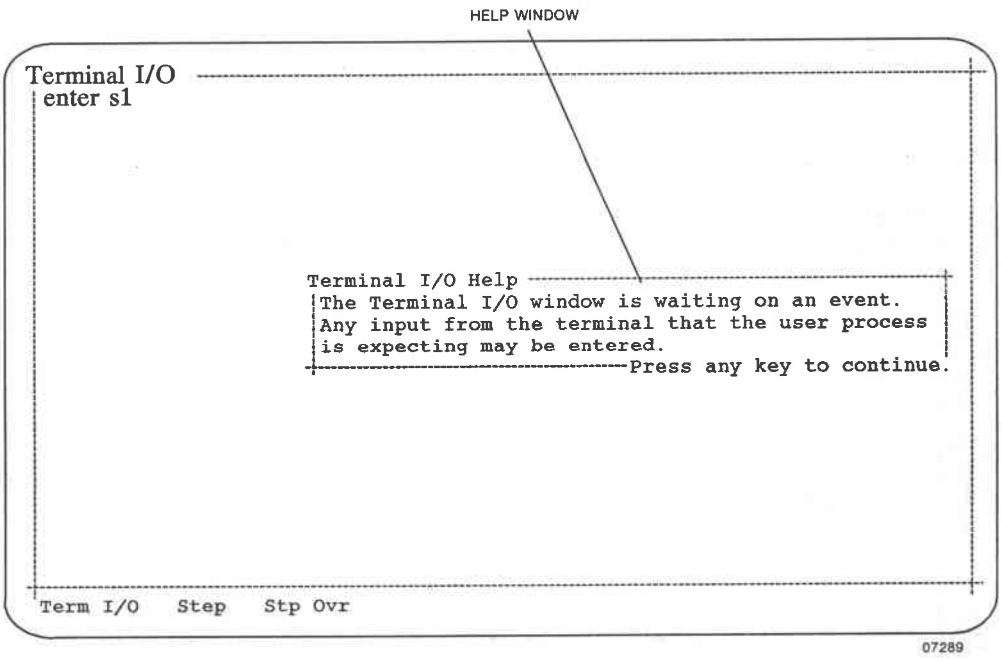
```
main() in "ask.c"
   1: #include <stdio.h>
   2:
   3: main()
 C 4: {
   5:       char    s1[80];
   6:
   7:       display(s1);
   8:       display(s1);
   9: }
   10:
                                              No process and no core file.
Command Output
>


                                                                  mdb.log
Command Input
*

 Term I/O   Step   Stp Ovr
```

Figure 1–5. Standard Screen with Open Log File

Menu Item 6: Terminal I/O Window
The Terminal I/O Window is toggled to enable or disable in the System Parameters Window (refer to Figure 1–4). When you enable the Terminal I/O Window *stdin, stdout,* and *stderr* are piped through the debugger program. When you disable the Terminal I/O Window, your user process has direct access to the terminal driver.

	enabled	disabled
user process I/O saved:	yes	no
control characters converted to readable format:	yes	no
stty(1) terminal setting for <EOF> processed:	no	yes

- The Terminal I/O Window is activated when the user process writes and flushes *stdout* or *stderr* or executes for approximately one second.

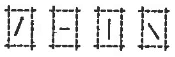


```
enter s1

Terminal I/O Help
The Terminal I/O window is waiting on an event.
Any input from the terminal that the user process
is expecting may be entered.
                                Press any key to continue.
```

Figure 1–6. Active Terminal I/O Window with Help Window

The Hour Glass Symbol

The **hour** glass symbol indicates that **mdb** is either in process or waiting for an event. When active, the symbol appears in the center of the screen (refer to Figure 1–7) and the frames move clockwise once a second, until the process is completed.



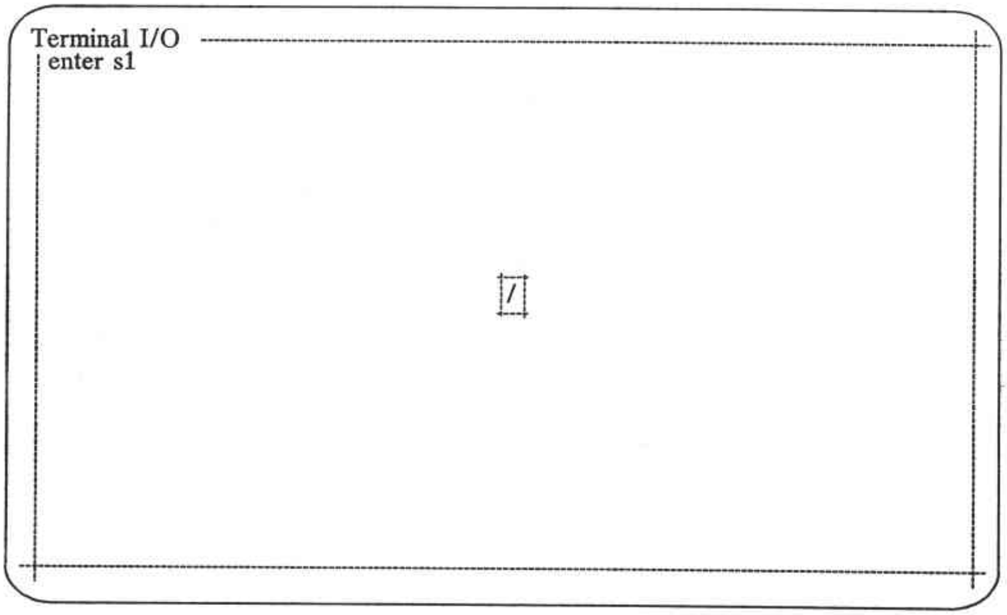
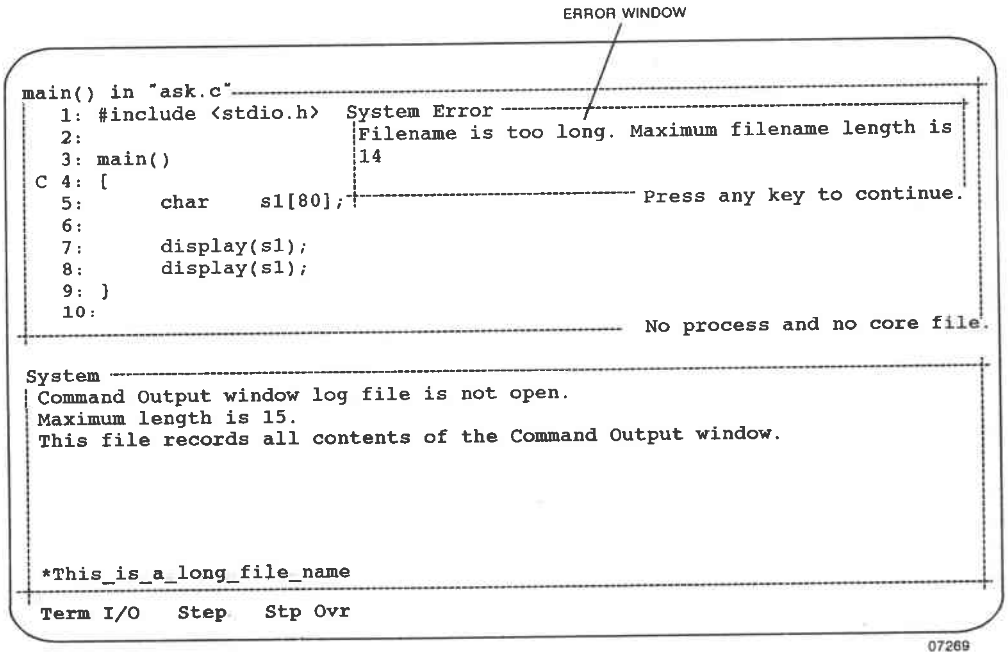


Figure 1–7. An Example of Screen with Hour Glass Symbol

The Error Window

The Error Window automatically displays all system and active window errors. Active window errors indicate an invalid action in the currently active window. Refer to Figure 1–8.



```
main() in "ask.c"
   1: #include <stdio.h>  System Error
   2:                     Filename is too long. Maximum filename length is
   3: main()              14
 C 4: {
   5:       char    s1[80];                      Press any key to continue.
   6:
   7:       display(s1);
   8:       display(s1);
   9: }
  10:
                                                 No process and no core file.
System
 Command Output window log file is not open.
 Maximum length is 15.
 This file records all contents of the Command Output window.




 *This_is_a_long_file_name
 Term I/O   Step   Stp Ovr
```

Figure 1–8. Active Error Window

Chapter 2

mdb Features

mdb permits full execution flow control. Breakpoints and single–stepping allow you to observe activity while the program executes. You can create more informative breakpoints and run–time patching by attaching commands to breakpoints.[1] Refer to Chapter 4, "Using mdb", for more information.

Breakpoints

Breakpoints let you examine program status by stopping program execution at specific locations and returning control to the user. When you set breakpoints and execute your program, execution continues until a break is encountered; at that point the word `>Breakpoint` is written to the Command Output Window and program execution is stopped. At this point you can interactively debug the program.

You can associate a command with a breakpoint by entering the command and breakpoint in the Command Input Window as follows:

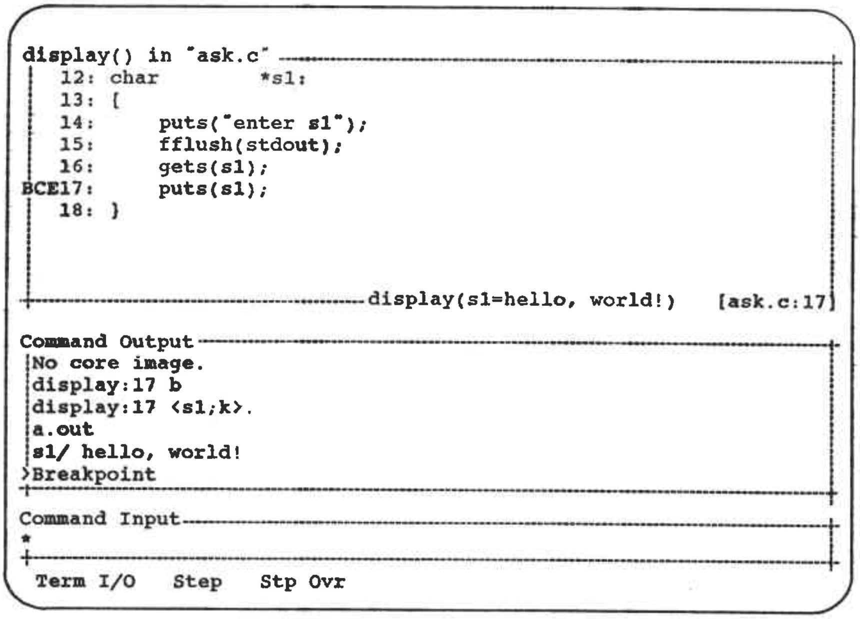
```
*17b s1;k
```

This breakpoint command is defined as follows:

`17`	go to line 17 in the current source file
`b`	insert a breakpoint command
`s1`	print the contents of the variable *s1*
`;`	separate the first command from the second command on the command line
`k`	stop execution

When you execute this command the contents of variable `s1` appear in the Command Output Window, followed by the word `>Breakpoint`. Refer to Figure 2–1.

[1] To fully utilize **mdb** execution flow control, the source program must be compiled using the **–g** option. The **–g** option causes the compiler to generate additional information about variables and statements in the compiled program. You can use **mdb** without compiling the source code with the **–g** option, but this limits the symbolic debugging capabilities.



```
display() in "ask.c"
    12: char        *sl;
    13: {
    14:     puts("enter sl");
    15:     fflush(stdout);
    16:     gets(sl);
BCE17:     puts(sl);
    18: }

                         display(sl=hello, world!)   [ask.c:17]
Command Output
No core image.
display:17 b
display:17 <sl;k>.
a.out
sl/ hello, world!
>Breakpoint
Command Input
*
```

07419

Figure 2–1. Command Output Screen after Breakpoint Executes

Single–Stepping

Single–stepping allows you to examine program status as you execute one line at a time or after you execute a specific number of lines.

The **s** command allows you to stop after each source line. The **S** command can 'step over' an entire procedure. In contrast to **s**, the **S** command executes a function without displaying the associated source code.

Command Files/Batch Mode

A command file allows you to execute one or more debugger commands without user interaction. This capability is sometimes referred to as batch mode operation. You can use command files to recreate a debugging session, implement automated test sequences, or redefine the screen environment without manually entering the command set.

The following example shows how the command file *setup.mdb* is used to redefine the logical size of the Command Output Window and open a log file.

```
^[ ^[ ^[ 448\n
 5mdb.log\n
 \n
 w\n
```

The command file is defined as follows:

`^[ ^[ ^[`	press the escape key three times to cycle to the System Window
`4`	select option number 4 from the menu
`48`	enter 48 as new value
`\n`	return to menu
`5`	select option number 5 from the menu
`mdblog.\n`	enter *mdb.log* as new file name and return to menu
`\n`	return to command input mode
`w\n`	redraw Source and Command Output Windows

Chapter 3

Installing mdb

The following sections tell how to install the GLS Symbolic Debugger (**mdb**) on the CLASSIC® Tri–Dimensional™ 97xx system. Before installing the debugger package you should be familiar with the **sysadm** installation tool. Refer to the **sysadm**(1M) manual page in your *REAL/IX Commands and Utilities Reference Manual* for a description of this tool.

Physical Requirements

The following list includes the minimum physical requirements for installing and using **mdb**:

- A CLASSIC Tri–Dimensional 97xx computer
- One 150 Mbyte cartridge tape drive
- One auxiliary input device for batch processing (any MODCOMP supported terminal)
- Approximately 300 Kbytes of disc space for **mdb** product files

Installation Tape Contents

The GLS Symbolic Debugger installation tape contains four IGF images (refer to **igf**(1M) in the *REAL/IX Commands and Utilities Reference Manual*). IGF image 3 contains the scripts required to install **mdb**. These files are removed when installation is complete. IGF image 4 contains all the product files required to install and support this product. See Table 3–1 for a breakdown of the IGF images.

Table 3–1. IGF Files/Image Contents

IGF FILES/IMAGE CONTENTS	TEMPORARY DIRECTORY*	INSTALLED DIRECTORY	DESCRIPTION
IGF IMAGE 1: Volume ID Header			
IGF IMAGE 2: Tape Directory			lists all directories
IGF IMAGE 3:			
README_mdb	*/usr/tmp/igfdir*	*/usr/mdb/README_mdb*	*README* file
INSTALL	*/usr/tmp/igfdir*		install script
UNINSTALL	*/usr/tmp/igfdir*		uninstall script
Rlist	*/usr/tmp/igfdir*		list of files removed during uninstall
IGF IMAGE 4:			
./bin/mdb		*/usr/bin/mdb*	mdb debug package
./catman/p_man/man1/mdb.1.z		*/usr/catman/p_man/man1/mdb.1.z*	mdb man page

* */usr/tmp/igfdir* is removed after installation is complete

Preinstallation

We advise that you remove any prior versions of this product before installing the new release. Follow the instructions in the section called "Removing mdb From Your System", at the end of this chapter.

Installation Procedure

Steps 1 through 6 tell you how to install **mdb**:

1. Log on to the operating system as superuser, also known as *root*.
2. Invoke the **sysadm**(1M) tool. You will be presented with the **sysadm** menu.
3. Select item **2** from the menu to display the *Software Management Menu*.
4. Choose item **1** from the menu to select *installpkg*.
5. Insert the installation tape into the cartridge tape drive and press carriage return (<CR>).
6. Choose item 2 to install a MODCOMP tape. The *INSTALL* script will complete the installation procedure. If any errors occur during the installation, a message describing the corrective action will be printed to the terminal.

After the installation, the README file resides in */usr/mdb/README_mdb*. You can remove this file from the system if desired.

Removing mdb From Your System

If you are updating your version of **mdb**, remove the older version before installing the new release. Use the cartridges tape that contains the <u>prior</u> release and follow these instructions <u>before</u> installing the new version. Follow the steps listed below to remove **mdb** from your system.

1. Log on to the operating system as superuser, also known as *root*.
2. Invoke the **sysadm**(1M) tool. You will be presented with the **sysadm** menu.
3. Select item **2** from the menu to display the *Software Management Menu*.
4. Choose item **3** from the menu to select *removepkg*.
5. Insert the <u>original</u> installation tape into the cartridge tape drive and press carriage return (<CR>). The *Rlist* file, which was originally removed after the installation procedure, is reinstalled and referenced by **sysadm**. The *UNINSTALL* script removes all **mdb** product files from your system. If an error occurs during this process, a message describing the corrective action is printed to the terminal screen.

Chapter 4

Using mdb

This chapter provides information about calling **mdb** and running **mdb** commands. The following conventions are use to help describe the **mdb** commands:

xxx	Italic letters represent a user-defined variable or variable file name, such as *core* or *a.out*.
xxx	Bold keywords are entered as shown.
[ ]	Brackets indicate an optional parameter.
...	An ellipsis indicates a continuing sequence.
...*n*	An ellipsis followed by the letter *n* indicates an indefinite sequence.

Calling mdb

Use the following syntax to call **mdb**:

mdb [**−w**] [**−W**] [*objfil* [*corfil* [*directory list*]]] [<*filename*]

−w	allows you to overwrite *objfil* locations
−W	suppresses **mdb** warnings produced when the source files that generate *objfil* cannot be found or are more recent than *objfil*
objfil	is an executable program file compiled with the **−g** compile option. The **−g** option causes the compiler to generate additional information about variables and statements in the compiled program. You can use **mdb** without compiling the source code with the **−g** option, but this limits the symbolic debugging capabilities. For more information about the **−g** compile option, refer to the *GLS Programming Guide*. The default object file is named *a.out*.
corfil	is a core image file produced after executing *objfil*. If you specify a dash (−) in place of *corfil*, **mdb** ignores any core image file. The default file name is *core*.
directory list	lists directories, separated by colons, used to locate the source files that build *objfil*.
<*filename*	is a command file. When *filename* is specified, commands are read from this file first. After EOF, command input is taken from the terminal.

Running mdb

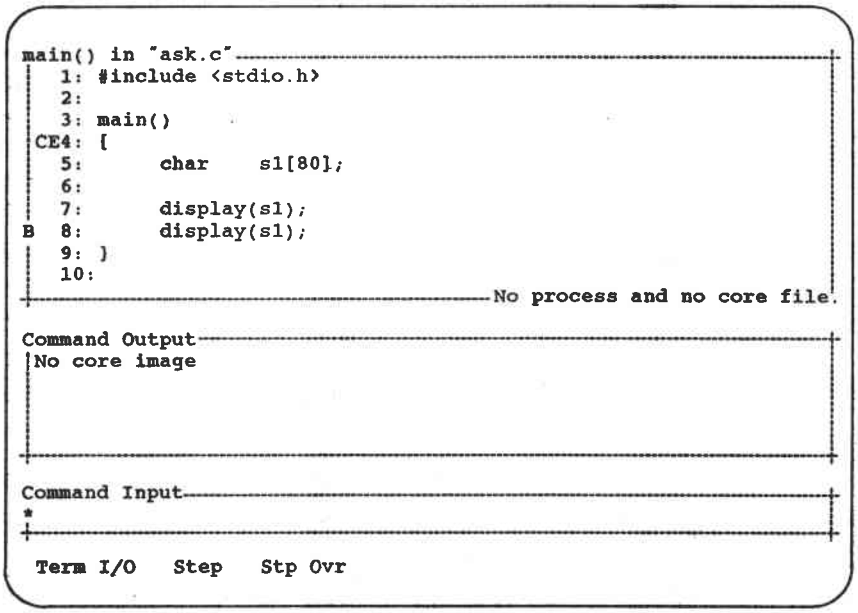
This section provides the following information about running **mdb**:

- current line and file
- addressing
- accessing variables

When you execute **mdb**, you are presented with the initial Standard Screen, which includes the Source, Command Output, and Command Input windows (see chapter 1).

Current Line and File, Last Line, and Breakpoints

At any given time there is a *current line* and *current file* in the Source Window. The source program you are examining is the current file; the current line, labeled with a `C`, tells you where the **mdb** cursor is located. The *last line*, labeled with an `E`, indicates the last line executed. A `B` labels a *breakpoint* line. Refer to Figure 4–1.



```
main() in "ask.c"-----------------------------------------------------+
|   1: #include <stdio.h>                                              |
|   2:                                                                 |
|   3: main()                                                          |
|CE4: {                                                                |
|   5:       char    s1[80];                                           |
|   6:                                                                 |
|   7:       display(s1);                                              |
B  8:       display(s1);                                               |
|   9: }                                                               |
|  10:                                                                 |
+--------------------------------------------No process and no core file.

Command Output--------------------------------------------------------+
|No core image                                                         |
|                                                                      |
|                                                                      |
|                                                                      |
+---------------------------------------------------------------------+

Command Input---------------------------------------------------------+
*                                                                      |
+---------------------------------------------------------------------+

 Term I/O   Step   Stp Ovr
```

07299

Figure 4–1. Current Line, Last Line, and Breakpoint Symbols

Normally the current line and file are set to the first line in *main*(). If you want to examine the current line and source statement at the point where the object program terminated, you can specify *corfil* when you call **mdb**. For more information about *corfil*, refer to the "Calling mdb" section in this chapter.

Addressing

File mapping determines how the address in a file is associated with a written address. Each mapping is represented by two triples (*b1*, *e1*, *f1*) and (*b2*, *e2*, *f2*). All addresses are kept as signed 32-bit integers. Mapping variables are defined as follows:

b1	beginning of instruction space
e1	end of instruction space
f1	offset into *a.out*
b2	beginning of data space
e2	end of data space
f1	offset into *core*

Calculate the file address that corresponds to a written address as follows:

b1 <= address < e1

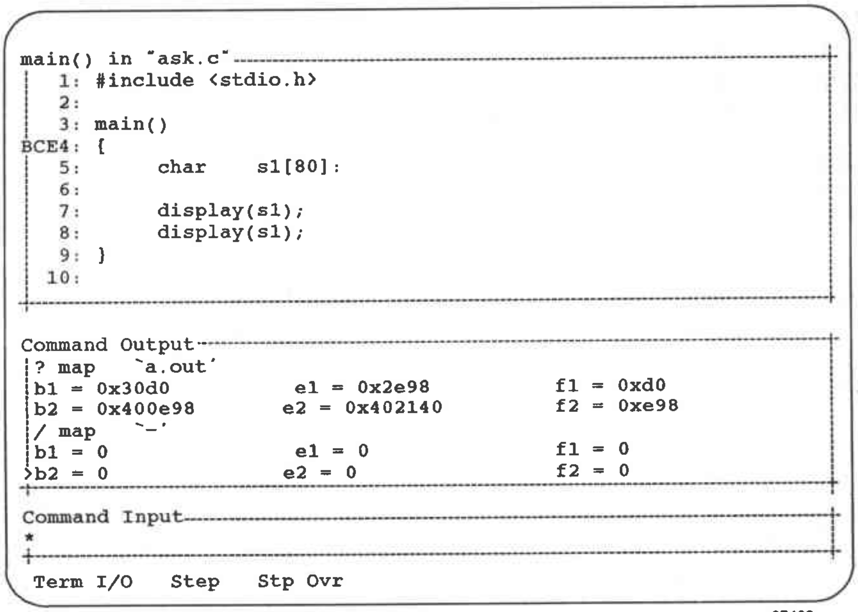
file address = address + f1 − b1

or

b2 <= address < e2

file address = address + f2 − b2

In programs with separate instruction and data space, the two segments for a file can overlap. Initially, both algorithms are suitable for typical *a.out* and *core* files. If either file is not in the standard format, then *b1* is 0, *e1* is the maximum file size, and *f1* is 0 for that file. This allows you to examine the complete file with no address translation.



```
main() in "ask.c"
    1: #include <stdio.h>
    2:
    3: main()
 BCE4: {
    5:      char    s1[80]:
    6:
    7:      display(s1);
    8:      display(s1);
    9: }
   10:

Command Output
 ? map   `a.out'
 b1 = 0x30d0            e1 = 0x2e98          f1 = 0xd0
 b2 = 0x400e98         e2 = 0x402140         f2 = 0xe98
 / map   `-'
 b1 = 0                 e1 = 0               f1 = 0
>b2 = 0                e2 = 0                f2 = 0

Command Input
 *

  Term I/O   Step   Stp Ovr
```

07429

Figure 4–2. An Example of File Mapping for a.out

Unless *objfil* or *corfil* are used to refer to a specific process, all process addresses refer to the currently executing program. Refer to Figure 4–2.

Accessing Variables

This section tells how to use **mdb** to access variables in your source program. **mdb** commands are described in the "mdb Commands" section in Chapter 5. The following list defines supported variable types and includes the appropriate syntax. These variable forms may be combined.

❑ local variables [*procedure*:]*variable*	accesses a variable by its procedure name. When you omit *procedure*, **mdb** defaults to the current procedure. To reference GLS FORTRAN common variables specify the common block name for *procedure*. You can specify the structure address in decimal, octal, or hexadecimal.
❑ structure members *variable.member*	specifies an individual structure element. The structure is interpreted as a set of variables and **mdb** displays the value of the specified element.
❑ pointers to structure members *variable -> member*	displays a member of the structure pointer. You can dereference pointers using *pointer*[0].
❑ common variables *common.variable*	allows you to access common variables by using the common block instead of the structure name. If you use a structure address rather than the structure variable name, it becomes the structure address and the last structure referenced by **mdb** becomes the structure template. Blank common is referenced by *.variable.*
❑ array elements *variable*[*number*]	provides specific index into the array.

- multidimensional array elements[1]

variable [*number*]... | *variable* [*number*,*number*...]	provides an actual index into the array.
variable [*]	indicates all legitimate values of that subscript. You can omit the last subscript. If all subscripts are omitted, **mdb** displays the entire array.
variable [*number*;*number*]	specifies a range of values.

- a particular variable on the stack

procedure:*variable*,*number*	specifies where the procedure resides on the stack, counting from the top. When you omit *procedure*, **mdb** uses the currently executing procedure.

- a line number in a source program

filename:*number* | *procedure*:*number*	specifies a line number by file name or procedure name. In either case, *number* is relative to the beginning of the file. If you omit *procedure* or *filename*, **mdb** uses the current file. If you omit *number*, **mdb** uses the first executable line of *procedure* or *filename*.

[1] You cannot display the contents of a multidimensional parameter as an array in a GLS FORTRAN program. The parameter points to the location of the array.

Chapter 5
mdb Commands

This chapter lists the **mdb** commands by the following functional groupings:

- examining data in a program
- examining source files
- controlling the execution of a source program
- miscellaneous commands

Examining Data in a Program

Use the following commands to examine data in your program:

t	Print a stack trace of the terminated or halted program.
T	If verbose mode is >=1, print the top line of the stack trace. This information is available in the bottom, right–hand corner of the source window.
variable [/[*clm*]]	Print the value of *variable* according to count *c*, length *l*, and format *m*. If you omit *c*, *l*, and *m*, **mdb** chooses a length and a format suitable for the variable type, as declared in the program. Although /, *c*, *l*, and *m* are optional parameters, the backslash (/) is required when you specify *c*, *l*, or *m* to identify them as variables, not commands. The last variable may be redisplayed with the command ./. *c* specifies the number of memory units to display, starting with the address implied by *variable*. *c* defaults to a special case when format commands **s** or **a** are used (refer to the *m* commands on the following page). *l* represents the output length of the value specified by *variable* (sometimes resulting in truncation). If *l* is not specified, the size comes from *variable*. The number of bytes in one unit of memory is determined by *l*. *l* can be **b** for one byte, **h** for two bytes (half word), or **l** for four bytes (long word). *l* is valid only when *m* is **c, d, u, o,** or **x**.

m represents one of the following commands used to format *variable*:

a	print characters starting at *variable's* address. You cannot use this format with register variables. If *c* is not specified, successive characters print until either 128 characters or a null byte is detected.
c	character
d	decimal
f	32–bit single precision floating–point
g	64–bit double precision floating–point
i	disassemble machine–language instruction with addresses printed numerically and symbolically
I	disassemble machine–language instruction with addresses printed numerically
o	octal
p	pointer to procedure
s	assume *variable* is a string pointer and print characters at the address pointed to by *variable.* If *c* is not specified, successive characters print until either 128 characters or a null byte is detected.
u	unsigned decimal
x	hexadecimal

The **sh**(1) metacharacters * and ? can be used within procedure and variable names as a device for pattern matching. If no procedure name is given, variables local to the current procedure and global variables are matched; if a procedure name is specified, then only variables local to that procedure are matched. To match only global variables, the form *:pattern* is used.

Example: When you issue the command,

```
argc/
```

the value of the variable argc appears in the Command Output Window.

linenumber | *variable*:[?*lm*]

Print the value at the address from *a.out* specified by *linenumber* or *variable* according to length *l* and format *m*. *variable* is the procedure name.

l can be one of the following: **b** for one byte, **h** for two bytes (half word), or **l** for four bytes (long word). *l* is valid only when *m* is **c**, **d**, **u**, **o**, or **x**.

m represents one of the following commands used to format *variable* (defaults to **i**):

a	print characters starting at *variable's* address. You cannot use this format with register variables.
c	character

d	decimal
f	32-bit single precision floating-point
g	64-bit double precision floating-point
i	disassemble machine-language instruction with addresses printed numerically and symbolically
I	disassemble machine-language instruction with addresses printed numerically
o	octal
p	pointer to procedure
s	assume *variable* is a string pointer and print characters at the address pointed to by *variable*
u	unsigned decimal
x	hexadecimal

Example: When you issue the command,

```
main:?
```

the value at the address specified by `main` appears in the Command Output Window.

variable[:*linenumber*]|*number* =[*lm*]

Print the address of *variable* or the value of *number* in the format specified by *lm*. Specify this command with *number* to convert between decimal, octal, and hexadecimal. The address of the *linenumber* in *variable* is specified by *variable:linenumber*.

l represents the output length of the value specified by *variable* (sometimes resulting in truncation). *l* can be one of the following: **b** for one byte, **h** for two bytes (half word), or **l** for four bytes (long word). The default for *l* is **l**. *l* is valid only when *m* is **c**, **d**, **u**, **o**, or **x**.

m represents the format used for displaying *variable* and can be one of the following (defaults to **x**):

a	print characters starting at *variable's* address. You cannot use this format with register variables.
c	character
d	decimal
f	32-bit single precision floating-point
g	64-bit double precision floating-point
i	disassemble machine-language instruction with addresses printed numerically and symbolically
I	disassemble machine-language instruction with addresses printed numerically
o	octal

p	pointer to procedure
s	assume *variable* is a string pointer and print characters at the address pointed to by *variable*
u	unsigned decimal
x	hexadecimal

Refer to the following example.

Example: When you issue the command,

```
argv[0]=
```

the address of the array element `argv[0]` appears in the Command Output Window.

variable!value

Assign *value* to *variable*. *variable* is a variable or an expression. When *variable* is an expression, it indicates that there may be more than one variable. For example, an array or structure is an example of a variable expression. If the address of *variable* is provided, **mdb** assumes it is integer.

value is a number, character constant, or a variable. *value* must be defined; expressions that produce more than one value, such as structures, are not allowed. **mdb** assumes registers and numbers are integers, unless the number contains a decimal point or exponent. Character constants are denoted as *'character*.

Use C programming language conventions for type conversions to perform the assignment. For more information about the C conventions, refer to the *GLS Programming Guide*.

Example: When you issue the command,

```
argc!2
```

2 is assigned to variable, `argc`, and the new value appears in the Command Output Window.

X | x

Print the current machine-language instruction. If you specify **x**, **mdb** also prints the machine registers.

Examining Source Files

Use the following commands to examine your source files:

e [*procedure* | *filename* | *directory/* | *directory filename*]

Set the current file to the *filename* or the file containing *procedure* or change the value of *directory*.

mdb sets the current line to the first line in *procedure* or *filename*. **mdb** assumes that the source files are in *directory*. The default for directory is the current working directory.

directory/ and *directory filename* change the directory pathname.

If verbose mode is >=1 and no parameters are specified, this command prints current procedure and file name. The current procedure and file name become the new source window header.

Example: When you issue the command,

```
e /usr/include/values.h
```

the directory path, `/usr/include/values.h`, becomes the current file and appears in the Command Output Window.

/ | **?***regular expression*[**/** | **?**]

The term *regular expression* refers to a character or character string that has special meaning when used to represent another character, string, line, or file name. *regular expressions* are used in context searching; they allow you to search for and operate on specified file characters, character strings, or lines using a single command. When you specify **/**, **mdb** searches forward. when you specify **?**, **mdb** searches backward. The trailing **/** or **?** is optional.

Example: When you issue the command,

```
/fflush
```

the *regular expression* **/** causes **mdb** to search forward for the specified character string, `fflush`. The line containing the matching string becomes the current line in the Source Window.

[*number*]**p** If *number* is specified, set the current line to *number*. If verbose mode is >=1, print the current line. This line is labeled by **C** in the Source Window.

[*number*]**z**	If *number* is specified, set the current line *number*. If verbose mode is >=1, print the current line followed by the next 9 lines. Set the current line to the last line printed. This updates the Source Window.
[*number*]**w**	If *number* is specified, set the current line number. If verbose mode is >=1, print the 10 lines around the current line. This line is labeled by **C** in the Source Window. The **w** command forces the Source Window and the Command Output Window to be redisplayed.
number	Set the current line to *number* and if verbose mode is >=1, print the new current line.

[*count*]**+** | **−** [**p**|**z**|**w**]

Change the current line by *count* and print the new current line. **+** increments the current line and **−** decrements the current line. If **p**, **z**, or **w** is specified, perform the associated command.

Controlling the Execution of a Source Program

Use the following commands to control the execution of your source program:

[*count*] **r** [*args*] | **R**

Run the program. *args* is one or more arguments. If you specify **r** and omit *args*, **mdb** uses the previous arguments. **R** runs the program with no arguments. An argument that begins with **<** or **>** redirects the standard input or output, respectively. *count* specifies the number of breakpoints to ignore.

[*linenumber*] **c** | **C** [*count*]

Continue after a breakpoint or interrupt. *count* specifies the number of breakpoints the program must encounter before it stops. *linenumber* causes **mdb** to place a temporary breakpoint at a specified line and continue executing. **mdb** deletes the breakpoint when the command completes. **C** reactivates the signal that caused the program to stop. **c** ignores the signal.

linenumber **g** [*count*]

Continue after the breakpoint line specified by *linenumber*. *count* specifies the number of breakpoints to ignore.

s | **S** [*count*] Single–step the number of lines specified by *count*. If *count* is omitted, **mdb** steps the program for one line. Use **S** to step over procedure calls.

i | **I** Single–step by one machine–language instruction. **I** reactivates the signal that caused the program to stop; **i** ignores the signal.

variable$ | *address*: **m** [*count*]

Single–step the specified number of lines until the location specified by *variable*$ or *address*: is modified with a new value. If *count* is omitted, the default is infinite. *variable*$ must be accessible from the current procedure.

[*level*] **v** Toggle verbose mode. This affects **T**, **e**, **P**, **z**, **w**, *number*, **+**, **–**, **s**, **S**, **b**, **i**, <NEWLINE>, and EOF character (^D). If *level* is omitted, **mdb** prints only the current source file and/or subroutine name when either changes. If *level* is >= 1, **mdb** prints source code lines before they are executed. If *level* is >= 2, **mdb** also prints the assembler statements. The **v** command turns verbose mode off if it is on. The **a** command turns verbose mode on.

k Kill the program being debugged when issued from the command line. Stop program execution when part of breakpoint command.

procedure([*arg* [, *arg*...]]) [/*m*]

Execute *procedure* with the given arguments. Your program must have been compiled with the **−g** compile option specified to use this command. The backslash (/) causes the return value, if there is one, to print. *arg* can be an integer, character, or string constant or the name of a variable accessible from the current procedure. *m* causes the value returned by *procedure* to be printed according to one of the following formats:

a	print characters starting at the variable's return value. You cannot use this format with register variables.
c	character
d	decimal (default)
f	32−bit single precision floating−point
g	64−bit double precision floating−point
i	disassemble machine−language instruction with addresses printed numerically and symbolically
I	disassemble machine−language instruction with addresses printed numerically
o	octal
p	pointer to *procedure*
s	assume the variable is a string pointer and print characters at the address pointed to by the variable
u	unsigned decimal
x	hexadecimal

Example: When you issue the command,

```
strlen(0x3ffffd8c)/
```

the value returned by the procedure, `strlen`, appears in the Command Output Window.

[*variable:*|*linenumber*] **b** [*command* [;*command*]...]

Set a breakpoint at the line specified by *linenumber*. If you specify a procedure name without a line number, **mdb** places a breakpoint at the first line in the procedure. If *linenumber* is omitted, **mdb** places the breakpoint at the current line.

command is any valid **mdb** command. **mdb** executes *command* when it encounters the breakpoint, then the source program resumes execution. If *command* is omitted, execution stops just before the breakpoint and control returns to **mdb**. If you specify **k** for *command*, control returns to **mdb** and execution of the source program stops.

	Example: When you issue the command, `main:7 b` a breakpoint is inserted at line `7` in the procedure `main`.
B	Print a list of the currently active breakpoints. Active breakpoints are labeled with a **B** in the Source Window (refer to Figure 4–1).
[*linenumber*] **d**	Delete the breakpoint at the line specified by *linenumber*. If *linenumber* is omitted, you can delete the breakpoints interactively. **mdb** prints the location of each breakpoint and reads a line from the command input. If the line begins with a **y** or **d, mdb** deletes the breakpoint.
D	Delete all breakpoints.
l	If verbose mode is >=1, print the last executed line. This line becomes the current line.
linenumber **a**	Set a breakpoint at the line specified by *linenumber* and print the last executed line. If *linenumber* references a stack and not a specific procedure in the stack, **mdb** prints the top line of the stack trace. Each time this command is issued verbose mode is turned off.

Miscellaneous Commands

Following is a list of miscellaneous commands:

!*command*	Interpret *command* with **sh**(1).
<NEWLINE>	Advance the current line by one and if verbose mode is >=1, print the new current line if the previous command printed a source line. If the previous command displayed a memory location, display the next memory location.
end-of-file character	If verbose mode is >=1, print the next 10 lines. The end-of-file character is usually ^D. This updates the Source Window.
<*filename*	Read commands from *filename* until end-of-file, and then accept commands from standard input. When you nest this command or specify < as a command in a file, input terminates at the first end-of-file encountered.
M	Print the address maps.
M[?/] [*] *b e f*	Record new value for the address map. The arguments ? and / specify the text and data maps, respectively. **mdb** changes the first segment of mapping (*b1, e1, f1*) unless your specify *. If * is specified, **mdb** changes the second segment of mapping (*b2, e2, f2*). If you omit *b*, *e*, or *f*, **mdb** does not change the other map parameters. For more information about the address map and map parameters, refer to the "Addressing" section in Chapter 4.
" *string*	Print *string*. **mdb** recognizes the C <ESC> sequences specified in the format \character. *character* is a non-numeric character.
q	Exit the debugger.
V	Print the version number.
Q	Print a list of procedures and files being debugged.

Appendix A

Error Messages

The **mdb** error messages in this list are self–explanatory:

```
Not enough memory. Memory may be freed by decreasing Command Input History or
Command Output scrolling size.
```

`Command line too long. Maximum filename length is` *n*`.`

`Filename is too long. Maximum filename length is` *n*`.`

`Printf(3S) failure. Unable to write` *'filename'*`.`

`Value out of range. Valid range is` *n* `through` *n*`.`

`Invalid key pressed`

`Key pressed not a digit`

`Not a valid filename`

'filename' `exists and is being appended.`

`Value not a positive integer`

Internal mesages indicate there is a problem with the debugger software. If you receive any of the following error messages, record the commands you were using and contact MODCOMP customer service. The appropriate telephone number for your location is listed in the preface of this manual.

```
Internal mdb error

Curses(3X) library failure

Puts(3S) failure

Signal(2) failure

System(3S) failure
```

INDEX

Please comment on the publication's completeness, accuracy, and readability. We also appreciate any general suggestions you may have to improve this publication.

If you found any errors in this publication, please specify the page number or include a copy of the page with your remarks.

Your comments will be promptly investigated and appropriate action will be taken.

☐ If you require a written answer please check this box and include your address below.

Comments: ______________________________

Manual Title ______________________________

Manual Order Number ____________________ Issue Date ____________

Name ____________________ Position ____________

Company ______________________________

Address ______________________________

____________________ Telephone (   ) ____________

MODCOMP
an AEG company

Please fold and tape.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 3624 FT. LAUDERDALE, FL 33309

POSTAGE WILL BE PAID BY ADDRESSEE

MODULAR COMPUTER SYSTEMS, INC.
1650 W. McNAB ROAD
P.O. BOX 6099
FT. LAUDERDALE, FL 33340–6099

Attention: TECHNICAL DOCUMENTATION, M.S. 880

MODCOMP, founded in 1970, is a worldwide supplier of high-performance, real-time computer systems, products, and services to the industrial automation, energy, transportation, scientific, and communications markets. MODCOMP is an AEG company.

Corporate Headquarters:
Modular Computer Systems, Inc.
1650 West McNab Road
P.O. Box 6099
Ft. Lauderdale, FL 33340-6099
Tel: (305) 974-1380
Twx: 510-956-9414

International Headquarters:
Modular Computer Services, Inc.
The Business Centre
Molly Millars Lane
Wokingham, Berkshire
RG11 2JQ, UK
Tel: 0734-786808, TLX: 851849149

Latin American-Caribbean
Headquarters:
Modular Computer Systems, Inc.,
1650 West McNab Road
P.O. Box 6099
Ft. Lauderdale, FL 33340-6099
Tel: (305) 977-1795, TLX: 3727852

Canadian Headquarters:
MODCOMP Canada, Ltd.,
400 Matheson Blvd. East, Unit 24
Mississauga, Ontario
Canada L4Z 1N8
Tel: (416) 890-0666
Fax: (416) 890-0266

Sales & Service Locations
Throughout the World