Megamax, Inc.
DEVELOPMENT SYSTEMS

LASER DB

Source-level and Assembly-level Debugger

For Use With Laser C

Atari ST

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Megamax, Inc. Printed in the United States of America.

Megamax, Inc makes no warranty of any kind with respect to this manual or the software described in this manual. The user assumes any risk as to the quality, performance, and accuracy or this product. In no event will Megamax, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the performance or use of this product.

Copyright © 1988, 1989 Megamax, Inc. All rights reserved.

LaserC and LaserDB are trademarks of Megamax, Inc.

Atari® and Atari ST® are trademarks of Atari, Inc.

GEM® is a trademark of Digital Research Corp.

UNIX® is a trademark of AT&T Information Systems.

Contents

1 • Introduction. 1

Requirements. 2

Checking versions. 3

Installation. 4

Conventions. 5

Text Entry. 7

2 • Source Preparation. 8

Compilation. 9

Linking. 10

Startup. 11

3 • LaserDB Expressions. 17

Current Line/Scope. 18

Identifiers. 19

Extended Names. 19

Literals. 22

Type Casting. 23

Memory Range Checksum. 24

Line Number Operator. 25

Predeclared Variables. 26

Contents

Processor Registers. 27
Predeclared Temporaries. 27
Printing Formats. 28
Default Printing Formats. 29

4 • Using LaserDB. 32
Push Button Menu Bar. 34
Window Usage. 35
Resizing. 35
Scrolling. 35
Source Window. 37
Source Mode. 37
Assembly Mode. 38
Window Display. 38
Setting Breakpoints. 39
Expression Window. 41
Register/Stack Window. 43

5 • Commands. 44
Options. 45
Journal. 47
Calls 48

Contents

Search. 49

Breakpoints. 55

Watches. 59

Execution. 61

- Go 61
- Trace. 61
- Next. 62
- Step. 62
- Return. 62
- Flip. 62
- reLoad. 63

Quitting. 64

6 • Sample Session. 65

• Keyboard Shortcuts. 80

CHAPTER 1
INTRODUCTION

LaserDB is a software development tool which helps to find logical programming errors by showing the correspondence between a C program's source code and it's executable image as it runs. LaserDB allows interaction with the executable program using names and lines defined in the source. These interactions include tracing program execution, setting breakpoints at the source-line level, and printing and setting program variables using C syntax (C expressions).

In addition to being a source-level debugger, LaserDB is also a full-featured assembly-level debugger. The disassembly of any GEM executable can be viewed and scrolled through. Breakpoints can be set on machine instructions, and registers and memory can be displayed, monitored, and set using special extensions to the C expression evaluator.

LaserDB's mouse and keyboard multi-window user interface is not exactly like that of typical Atari applications, but was written such that it does not utilize the GEM-level routines of the ST's ROM — all window and dialog functions were written especially for LaserDB. It was necessary to insulate the debugger from the operating system as much as possible so that a buggy or ill behaved target program, perhaps one which calls GEM incorrectly, is not as likely to affect the debugger's operation. Another advantage, for those who develop for color monitors, is that LaserDB can debug programs which must operate in low-resolution, as games and graphic programs often do. LaserDB automatically switches to medium resolution for it's own display and then back to the target's resolution for it's screen. This is something which GEM was not designed to do.

Requirements

LaserDB can be run on Atari ST computers which have as little as 512Kb of RAM. However, because so much data must be in memory during a debug session, it is suggested that a system with 1Mb or more of RAM be used for source-level debugging. There are several ways, discussed throughout this manual, which can help to conserve precious memory.

Checking Version Numbers

The LaserC compiler and linker versions 2.0 or greater are capable of generating debugger information which is usable by LaserDB. To check the version of the LaserC compiler or linker (or any other command line utility), run it from the STDIO window with the -V flag. When used with no other options, -V will print copyright and version information. For example, to print the version of the LaserC compiler, type the following into the Laser. Shell's STDIO window:

```
CCOM.TTP -V <ENTER>
```

Note that <ENTER> refers to the “Enter” key on the keypad, not the “Return” key.

The LaserC Development System version 2.0 or greater is required to use LaserDB's source-level debugging features. The compiler and linker provided with version 1.01 of LaserC do not support debugger information compatible with LaserDB. For assembly-level debugging, any compiler or assembler can be used. If the linker used has the ability to include GEM style symbol information for global labels into the executable, LaserDB will use them when showing the disassembled code.

Installation

LaserDB can be installed on a hard disk by copying the file LDB.PRG into any folder. There are no restrictions as to the location of LaserDB with respect to the compiler or source files. LaserDB uses a configuration file called LDB.CFG. This file will be automatically created in the current working directory if it is not found.

Note

Before using LaserDB from a floppy disk, make a working copy of the distribution disk and store the original in a safe place.

When developing under the Laser Shell, it may be convenient to make LaserDB a tool so that it can be run simply by choosing it from the Execute menu. Because of the way in which LaserDB was created, it **can not** be made a RAM resident tool under the Laser Shell (check the LaserC manual for instructions on adding tools). If there is not sufficient RAM to debug a program from the Laser Shell, LaserDB can be run from the GEM desktop by double-clicking on LDB.PRG.

Conventions

The following conventions are used in this manual:

Cursor	The mouse pointer.
Insertion point	The typing cursor that appears in text entry items. Any typing is inserted at this point.
Press/type	Keyboard entry.
Click	The mouse cursor is positioned over an item and either the left or right mouse button is pressed and released.
Drag	The mouse cursor is positioned over an item and either mouse button is pressed and held. Now the item may be repositioned by moving the mouse (an outline of the item will follow the mouse movement). When the item has been repositioned, the mouse button is released.
Choose	Menu items are chosen by clicking on items or by pressing keys corresponding to the uppercase letters in each menu

	item.
Select	Non-menu items are selected with a click. Dialog items such as radio buttons are capable of being selected. The term "Select…" refers to buttons while "Choose…" refers to menu items.

Note

This manual assumes that the reader is familiar with the C programming language and the LaserC language implementation. Some parts of this manual assume familiarity with Motorola MC68000 assembly language programming and the in-line assembler, but these sections may be ignored if only source-level debugging is required.

Text Entry

Text entry is often required, such as when entering C expressions to print variables. Text entry occurs in editable text lines, such as the one pictured below.

Typing is added at the insertion point, which may be moved left or right with the arrow keys. The <Backspace> key erases the character to the left of the insertion point and the <Delete> key erases the character to the right of the insertion point. The <Esc> key erases the entire line. The entry line may or may not contain the '_' characters as place holders as shown above, depending on where it is used. The expression window entry line does not use '_' while dialog box text entries do.

CHAPTER 2
SOURCE PREPARATION

The correspondence between executable code and C source code is drawn by having the compiler include special debugger information into the object file. This information tells LaserDB where C source lines begin in the executable program, what the addresses of global variables and functions are, and what stack-frame offsets or registers are used for local variables. In addition, debugger information provides the types of variables, structure and union field names, enumeration values, and lexical scope information.

Using this information, LaserDB can hilight the currently executing source line, evaluate C expressions using the names defined in the source, and perform other related functions.

LaserDB can debug programs composed of multiple source files. The source files are concatenated (in memory) when they are read by the debugger, so that each has a unique range of line numbers. Because C source is related to it's executable program via source line numbers, it is best to place C statements on individual lines in the source files. Files which are included with the "#include" preprocessor command may not be debugged or viewed. Also, no macro definitions are available to the debugger.

Compilation

Depending on the desired method of development, one of two ways to generate debugger information from source files may be chosen.

The Compile dialog under the **Execute** menu of the Laser Shell version 2.x contains a check box to "Generate LaserDB information". This item should be checked before clicking on the OK button.

The command line -Z flag is accepted by the C compiler CCOM.TTP, and by the compile and link utility CC.TTP. This option causes the compiler to include debugger information in the object file.

Note

It is not necessary to generate debugger information for all component source files of a project. Source-level information can be large and can use up a significant amount of memory, especially while linking. Also during a debug session, the C source files *as well as the debugger information* for each file compiled for debugging is loaded into RAM. For these reasons, it may be a good idea to be selective about which source files to compile with debugger information and to compile the rest without it. To remove debugger information from an object file, simply recompile it's source without including debugger information.

Linking

As with compilation, there are two methods of linking with debugger information. Debugger information is included in the executable in such a way that the resultant program can be run as a stand-alone application. It will however be much larger than usual and should be relinked without debugger information to produce a final version of program.

The link dialog used by the Laser Shell version 2.x contains a check box to "Include LaserDB information in

executable". In addition to including debugger information, GEM style symbols will also be included in the executable for assembly-level debugging.

The -Z flag is also recognized by the linker LD.TTP when run from a command line. This option causes the linker to include debugger information in the executable program. As with the link dialog, the -Z option also tells the linker to include GEM symbols for all global names defined in the executable.

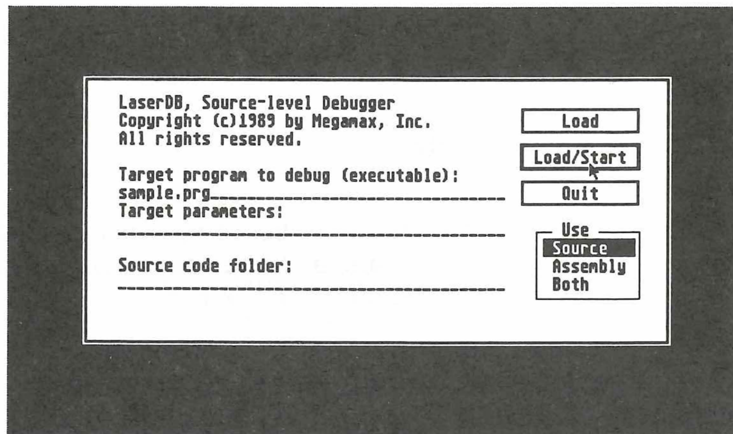
Note

Programs may be compiled and linked from a command-line shell. The Laser CC.TTP, compile and link utility, supports the -Z flag.

Startup

Once a program has been successfully compiled and linked with debugger information, LaserDB can be started. To start debugging from the desktop, double click on LDB.PRG. To start debugging from the Laser Shell, use the Execute menu to run LDB.PRG.

LaserDB provides a dialog box at startup in which the user specifies the executable program to debug along with other startup information. Provided the standard LaserC initialization code is included during the link, I/O redirection commands may be supplied to the target. This redirection is passed to the target program and ignored by LaserDB. I/O redirection can be specified on the command line if LaserDB is started from a command line shell.



When started, LaserDB first looks for it's configuration file LDB.CFG in the current working directory (usually this is the same folder in which LDB.PRG resides). If found, it restores all user settings from the last time the

debugger was terminated. This user information includes all settings in the Startup dialog as well as window positions. If the configuration file is not found, default settings are applied and a configuration file is created in the current working directory.

As pictured above, the dialog contains three editable text lines. The first *must* contain the executable program to debug. The file name entered here will be that output by the linker, and will usually end with .PRG or .TTP. Do not give a source .C or object .O file name here. The second line may contain the optional command line arguments to "main()". The third line may contain the folder in which and source files are to be found when debugging in the source mode.

The radio buttons on the right hand side of the dialog determine the desired debugging mode(s).

Source	Allow only source-level debugging. All source files which were compiled with debugger information will be loaded, and all source-level information for these files will be available. Source files are loaded from the folder specified in the "Source folder" item described below.

Assembly	Load only disassembly information and GEM style symbols, if available. GEM symbols are used by the disassembler to make the output more readable.
Both	Load both source-level information and assembly-level information. If this button is selected, the mode can be changed back and forth between source and assembly during debugging.

When the dialog is set as desired, the "Load" or "Load/Start" button will initiate the startup sequence.

Load	Loads debugger information according to the "Use" setting. The executable program is loaded and the program counter is stopped *before executing the first instruction*. If debugging at the source level, this will mean that no execution will begin until initiated from the **Execute** menu. If the function

	"main()" was compiled with debugger information, a **Step** will be required to begin single-stepping or tracing.
Load/Start	Loads debugger information as specified in the "Use" setting, and then loads the executable program and performs a **Step** to enter "main()". This button should be used when it is desired that the target run until the first source line, as when the file containing "main()" was compiled with debugger information.
Quit	Will leave the debugger.

After the configuration file is read, the target's component source files are loaded. Each source is loaded from the folder given in the startup dialog. If no folder is given, the current working directory is assumed. If a source file is not found, a message is displayed and the source and the other debugger information for that particular file is ignored. If assembly level debugging is specified, the entire executable is read and disassembled to

generate instruction address information. This information is stored in a very compact way in memory, but if no assembly-level debugging is required this option should not be checked. Finally the target executable is loaded and the debugger is ready go.

Note

If the "Load/Start" button is pressed, the program will execute until the first source line is reached — this will be the first function encountered whose source file was compiled with debugger information, and will only be the function "main()" if the file containing "main()" was compiled with debugger information.

CHAPTER 3
LASERDB EXPRESSIONS

Expressions are used extensively in LaserDB, most notably to print and set variables. The language accepted by the debugger's expression evaluator is that of the C language with some changes designed to facilitate debugging.

Several extensions have been added to the expression evaluator to make source-level debugging easier and to facilitate assembly-level debugging. All debugger functions which use expressions, including watches, conditional breaks, and the expression window, work identically in either source or assembly mode.

Current Line/ Scope

The **current line** is the source line or machine instruction upon which the program counter rests. It is the next line to be executed when the target is single stepped, traced, or run. LaserDB indicates the current line in the source window by hilighting it. In the source mode, the current source line will be hilighted. Not all source lines will contain code, as do comments and variable declarations. Local initializer lines will have code. The current line can be found by pressing the <Enter> key on the keypad.

The **current scope** is the lexical scope of the current statement as defined in the source. Lexical scope refers to the nesting of variable definitions in files, functions, or blocks. Expressions are always evaluated with regard to the current scope.

Identifiers

All names defined in the target's source are accessible to the expression evaluator, provided the source was compiled with debugger information. Names are always used in expressions exactly as they are declared in the source code — all characters in a name are significant.

GEM style symbols are included for all global and static names, even for those files which do not contain debugger information.

Note

The C compiler adds a leading underscore '_' to each global name and a leading '~' to each static name included in the GEM style symbols. Also note that GEM symbols are limited to a length of eight characters.

Extended Names

The lexical scope rules of C apply according to the current scope. For example, the program below contains two declarations of the name "i".

```
int         i;     /* global declaration of i */
main()
    {
    func();
    }

func()
    {
    int    i;     /* local declaration of i */
    i++;
    }
```

If program execution is halted (i.e. with a breakpoint) in the function "main()", any use of "i" in an expression would refer to the global declaration. If, however, the program were halted in the function "func()", "i" would refer to the local declaration.

A special naming convention has been incorporated to allow access to global and static variables which are hidden by local declarations in the current scope. The syntax for an extended name is:

```
\<variable>
```

For example, if a variable "count" is global, "\count" will refer to the global even if it is hidden by a local variable

"count" in the current scope. In another example, a static variable "node" is declared outside of a function. Using "\node" will supersede any local declaration of "node" *only* when the current scope is in the file which contains the static declaration for "node".

Note

When an expression contains a variable which is out of scope, the printed result of the expression will be "Out of scope". When a variable is not found in the symbol table, "Undefined" is printed.

Literals

The following literals are recognized by LaserDB's lexical analyzer:

Literal	Sample	
Integer	100	- decimal
	0100	- octal
	0x100	- hexadecimal
Floating-point	100.0	
Character	'a'	
escapes	'\r'	- return
	'\n'	- newline
	'\t'	- tab
	'\b'	- backspace
	'\v'	- vertical tab
	'\\'	- backslash
	'\''	- single quote
	'\"'	- double quote
	'\digits'	- octal constant
String	"Hello, world\n"	

Type Casting

The expression evaluator allows type casting to any predefined C type; “char”, “short”, “int”, “long”, “unsigned”, “float”, and “double” are all legal in cast operations. Pointer types are also allowed using the “*” constructor. For example, the following is a valid cast operator:

```
(char**)0x2044
```

In addition, casts to type defined names and structure tag names are also supported. Casts to aggregate types like structures and arrays are not supported.

The expression evaluator does not support function calling, structure passing, or “#define” macro definitions.

Memory Range Checksum

A memory range checksum may be specified by placing a colon between two addresses, representing a start and end. The syntax is:

```
<start expression> : <end expression>
```

The start and end may be any expressions resulting in pointers, and the result of the expression is type "int". The result is an integer checksum of all bytes from the start address up to the ending address. For example the expression:

```
array : array + 100
```

will perform a checksum on the first 100 elements of "array". This operator is useful for watching a range of memory or to compare two blocks of memory. The checksum operator has a precedence lower than the C operators and associates from left to right.

Line Number Operator

Another extension to the expression syntax is the line number operator. The unary '@' operator interprets an expression result as a line number in the source window, and converts it into an address. If no code is associated with a given line number, the closest match in the source window is found. If the beginning of the file is reached and no address found, an error is reported. Note that the address found depends on the mode. In the source mode; only source line starts will be found, where as in the assembly mode, machine instruction addresses will be found. The '@' operator is the same precedence as the memory checksum operator ':' and is right associative.

Predeclared Variables

The debugger maintains a set of internal variables which are not a part of the target program's variable space. These predeclared variables all start with a '$' character.

Predeclared variables are:

$text	The base address of the target's text (code) segment. The type is "int *".
$data	The base address of the target's data segment (initialized global and static variables). The type is "char *"
$bss	The base address of the target's bss segment (uninitialized globals and static variables). The type is "char *."
$end	The end of the bss segment. This is normally the extent of the program in memory. The type is "char *".
$line	The line number in the source window of the current statement or instruction. The type is "long".

Processor Registers

The MC68000 register names "$D0", "$D1", ... , "$D7" and "$A0", "$A1", ... , "$A7" are recognized in upper case only. The names "$PC", "$USP", "$SSP", and "$SR" are recognized in upper case only and represent the program counter, the user stack pointer, the supervisor stack pointer, and the status register, respectively. The data registers are type "long", the address registers are type "char *", the "PC" and stack pointers are type "int *", and the status register is type "unsigned int".

Predeclared Temporaries

There are 10 predeclared variables which can be used as temporaries during debugging. They are "$0" — "$9" and are all of type "long". These are useful for storing intermediate expression results or for creating pass counts for watch or break expressions (see chapter 5, page 44).

Printing Formats

Any expression may be optionally preceded by a format specifier. Formats are similar to the percent (%) conversions accepted by the C library function "printf()." They are used to control the way in which an expression result is printed. The full syntax of a format is:

```
%<format> [,] <expression>
```

That is, a '%' followed by a legal format character, followed by white space or and optional ',' followed by a debugger expression. For example, the null-terminated character string "str" can be printed with a "%s":

```
%s str
```

Notice that without the "%s" format, the pointer value of "str" will be printed. The allowed formats are:

Format	Meaning
%s	Print the following expression as a C style (null terminated) string.
%c	Print the following as an ASCII character.

%d	Print the following as decimal short, int, or long. Note that "%ld" is not needed to print a long, "%d" works for any representation.
%o	Print as an octal short, int, or long.
%x	Print as a hexadecimal short, int, or long.

Formats require no quotes or parenthesis but must appear before the expression and, either white space or a comma must separate the format from the rest of the expression. Some expressions are used by commands that do not print a result, such as breakpoint expressions. In these instances, any format will be ignored.

Default Printing Formats

If no format is specified, a type is chosen based on the resultant type of the expression. Types "short", "int", "long", and "unsigned" are one of "%d", "%o", or "%x", according to the default format setting (see Options). Character type defaults to "%c". All pointers are printed in hexadecimal. ASCII character strings can be printed with "%s", otherwise the value of the pointer will be printed.

Structures are a special case. If the result of an expression is a structure, the entire structure is printed, showing field names and value.

Operators

The operators recognized by the expression evaluator in order of precedence are:

```
primary
16          name literal
            $name                   predeclared name
            []                      subscripting
            .                       direct selection
            ->                      indirect selection

unary
15          ++ --                   postfix
14          ++ --                   prefix
            * & - ! ~

binary and ternary
13L         * / %                   multiplicative
12L         + -                     additive
11L         >> <<                   shift
10L         < > <= >=               inequality
9L          == !=                   equality/inequality
8L          &                       bitwise and
7L          ^                       bitwise xor
6L          |                       bitwise or
5L          &&                      logical and
4L          ||                      logical or
3R          ? :                     conditional
2R          = += -= *= /=           assignment
            ^= %= &= |=
            >>= <<=
1L          ,                       sequential evaluation

special
0L          :                       memory range checksum
0R          @                       line number
```

CHAPTER 4
USING LASERDB

The LaserDB screen is divided into window tiles, each responsible for displaying different information. A window tile has a display area, and may have a scroll bar along the right hand side. At the top of the screen is a menu bar and at the bottom of the screen is a message line.

LaserDB uses the Atari's alternate screen mechanism for it's display. The debugger maintains it's own screen which occupies separate memory from that of the application. This means that the debugger will not overwrite the target program's screen, so that each can alternately be viewed while debugging.

The debugger is controlled by choosing menu items or by interacting in some way with the display. There are nu-

merous keyboard "shortcuts" to help cut down on hand movement between the keyboard and the mouse. These shortcuts are summarized in the appendix.

Push Button Menu Bar

The menu bar located at the top of the screen does not use the typical drop-down menus used by GEM. Menu commands can be chosen by typing the first letter of the desired command as displayed in menu, or with the mouse by clicking on the desired command in the menu bar.

Some menu items perform an action while others change the commands in the menu bar. These sub-menus can be canceled by choosing the "<Esc>" menu item or by pressing the <Esc> key.

```
Command/Execute: Go  Trace  Next  S
 Watches

 [B] Source: sample
   17      printf
   18
```

Window Usage

Up to four windows may be simultaneously visible on the screen. The top window contains watch expressions — debugger expressions which are monitored during program execution. The middle window contains the target program's source code or the disassembly of the target executable, depending on the mode setting. The expression window, located at the bottom of the screen, is where selected expressions are evaluated to print or set variables. The register window on the right hand side of the screen may be displayed or hidden as desired.

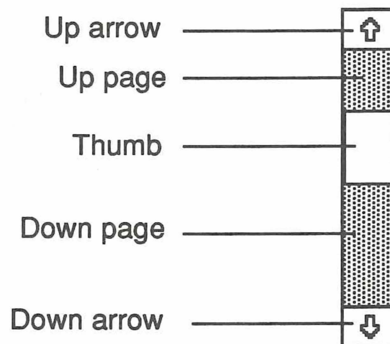
Resizing

Windows may be resized horizontally as desired to show more or less of a particular window's contents by dragging up or down the title bars which lie between two windows. By decreasing the size of one window, the above window is made larger. Note that the watch window title cannot be dragged.

Scrolling

Often a window contains more lines of text than can be displayed in the content area. In this case, scroll bars are used to position a window over it's contents. Not only can scroll bars be manipulated with the mouse, but there are also keyboard shortcuts for line, page, and home/end scrolling of the source window.

A scroll bar is composed of two opposing arrows, a page scroll area, and a thumb. The thumb changes size according to the number of lines visible in the window.



Clicking on an arrow moves the window over it's contents in the indicated direction by one line. Clicking in the page scroll area moves the window by one page. The thumb may be dragged to reposition the window over any part of it's contents.

Source Window

LaserDB operates in one of two modes; source or assembly. The mode may be changed at any time during a debug session with the Options command, provided both source and assembly information is present as specified in the Startup dialog. The mode may also be changed by pressing "<Ctl>-M".

```
Command: Control  Search  Execute  Breaks  Watches  Quit
 Watches

 [0] Source: sample1.c
  10
  11 /*  The function main()
  12 */
  13 main()
  14     {
  15     register int        i;
  16
  17     printf("This is a sample\n");
  18
  19     for ( i = 0; i < 1000; i++ ) {
  20         l   = i - 10;
  21         f   = l + 2.5;
  22         }
 Expressions - <Tab> to enter
```

Source Mode

In the source mode, only the target C code is displayed in the source window. Program single stepping, tracing, and breakpoint setting occur by source lines. In-line assembly is shown as source, and each "asm { }" is treated as a single statement.

```
Command/Execute: Go  Trace  Next  Step  Return  Flip  reLoad  <Esc>
 Watches                                                        Regs.
                                                             ⇧ D0 0000000A
                                                               D1 00000003
                                                             ⇩ D2 00000000
 [B] Assembly: TEXT Segment                                    D3 00000000
 00042D62         BLT.S     0x00042D22                       ⇧ D4 00000000
 00042D64         CLR.W     D7                                 D5 00000000
 00042D66         BRA.W     0x00042D8E                         D6 00000000
 00042D6A         MOVE.W    D7,-(A7)                           D7 00000032
 00042D6C         JSR       _func1.L                           A0 000475DA
 00042D72         ADDQ.L    #2,A7                              A1 00049738
 00042D74         MOVE.W    D7,D0                              A2 00000000
 00042D76         ADDQ.W    #2,D0                              A3 00000000
 00042D78         MOVE.W    D0,-(A7)                           A4 000456F6
 00042D7A         MOVE.W    D7,D0                              A5 00042B4A
 00042D7C         ADDQ.W    #1,D0                              A6 00047866
 00042D7E         MOVE.W    D0,-(A7)                         ⇩ US 00047862
 Expressions - <Tab> to enter                                  SS 00007554
-7.500000                                                    ⇧
> %x 1                                                         PC 00042D6A
0xFFFFFFF6                                                     CCR xNzvC
> i=0                                                              User
0                                                            ⇩ Mask: 3
 Breakpoint encountered
```

Assembly Mode

In the assembly mode, the target executable is disassembled into the MC68000 instructions output by the compiler or in-line assembler. Program stepping or tracing may be performed at the assembly level.

Window Display

The source window contains the concatenation of source files in the order in which they were supplied to the linker. Since the order in which object files are linked determines the order in which code and data are concatenated into the executable, the source window accurately depicts the target's structure. The title bar shows the source file name which is currently displayed in the

window.

The window may be scrolled with the scroll bar, or from the keyboard using the up and down arrows to scroll by lines, <Shift>-up and down arrows to scroll by pages, and <Home> and <Shift>-<Home> to scroll to the top and bottom, respectively, of the window.

There are several indicators associated with the source window. Line numbers may be shown before each line of source. The current source line, or the current machine instruction in the assembly mode, is hilighted. If the program counter lies outside of the source window, the hilight will be on the very first line or the very last line in the window. The current source line or machine instruction can be brought into view by pressing the <Enter> key on the keypad.

tting Breakpoints With The Mouse

Breakpoints can be set on source lines or machine instructions by simply clicking on the desired line. Lines which have breakpoints are indicated in the source window by a special character, printed at the beginning of the line. Clicking on a line which already has a breakpoint removes the break. The '*' key on the keypad toggles a breakpoint on the current line, and the '/' character on the keypad toggles a breakpoint on the top line

of the window. This is useful for searching and then setting a breakpoint on what was found, since it will appear on the first line. A maximum of 20 breakpoints may be set at one time. The window title contains an indictor "[B]" when breakpoints are enabled, or a "[-]" when breakpoints are disabled. For more information on Breakpoints see page 55.

Expression Window

It is in the expression window that variables are printed. The <Tab> key is used to initiate an expression. When the <Tab> key is typed, a text entry line will appear in the expression window. Any legal debugger expression may be entered (see chapter 3). Once entered, the <Return>, <Enter>, or <Tab> key will evaluate the expression.

To print a variable, simply type the following and then press <Return>:

```
variable
```

To assign a value to a variable type:

```
variable = value
```

Any valid expression can be printed. The command:

```
%s variable + 5
```

will add 5 to the value of variable, then use it as an address to print a null-terminated string, and

```
$D0 = *((char*)$A1 + 10)
```

will cast the value of address register “A1” to a character pointer, add 10 to this value, and then dereference and assign the character to data register “D0”.

The expression window may be resized and scrolled with the mouse. Scrolling up reveals a history of the last 20 evaluated expressions and their results. Scrolling may be done with the scroll bar or by using the ‘+’ and ‘-’ keys on the keypad while not entering an expression. Clicking the mouse in the window's content area is the same as typing <Tab> to start an expression.

Often, the same variable or expression is printed repeatedly during execution of the target. To facilitate this, the up and down arrows can be used *while entering an expression* to copy previous expressions into the current line. Each time the up arrow is pressed, the previous expression is copied. The down arrow reverses the history.

Note

In addition to the expression history, the Function keys can be set to often-used expressions for easy entry. See the section on Options for more information

Register/Stack Window

The columnar window on the right of the screen displays either the MC68000 registers or a hexadecimal dump of the user stack. The stack dump shows words at positive offsets from the current stack pointer, either US (user stack) or SS (supervisor stack). In addition, the current program counter "PC", the condition code register "CCR", the interrupt mask, and the current mode are shown. Condition code bits which are set are in upper case.

Clicking anywhere on the register window toggles between register and stack display. The keyboard shortcut "<Ctl>-R" toggles the register window on and off.

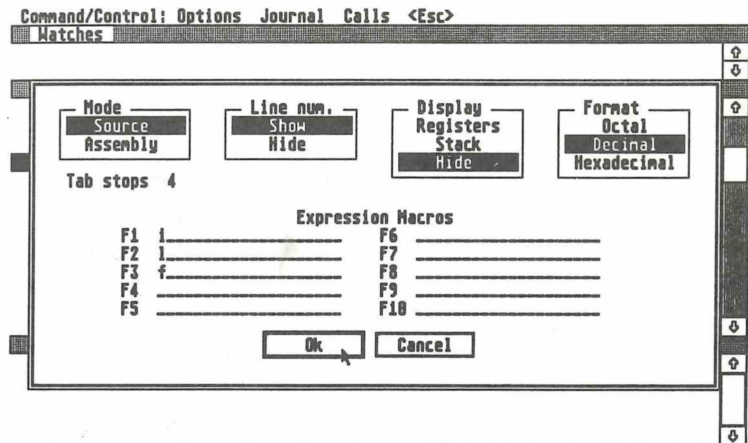
CHAPTER 5
COMMANDS

This chapter describes in detail the LaserDB menu commands, giving examples of how they are useful for debugging.

Dialog boxes may contain push buttons, radio buttons, editable text lines, and scrolling regions. The mouse is required to select push buttons, radio buttons, and to scroll a region. Push buttons which have a thick border can be selected with the <Return> key. The left and right arrows as well as the mouse can be used to reposition the insertion point in an editable text line. The up and down arrow keys, the <Tab> key, and the mouse can be used to move the insertion point among text edit lines in a dialog.

Options

The **Options** command under the **Control** menu brings up a dialog which controls LaserDB user settings. When chosen the dialog shows all current settings. All items in this dialog are saved in the configuration file.



Options are:

Mode	Source or Assembly mode (see page 32).
Line Numbers	Show or hide line numbers before each line in the source window.

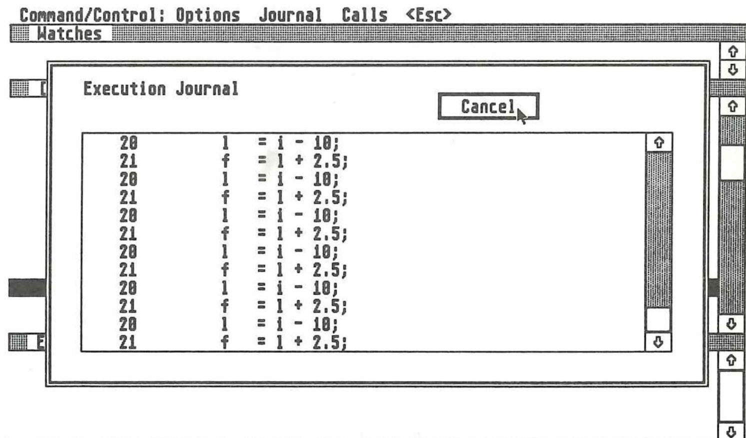
Registers	Shows registers, stack dump, or hides the window.
Format	Determines the default format for printing expression results of integer types. “Octal” will use “%o”, “Decimal” will use “%d” and “Hexadecimal” will use “%x”.
Tab stops	This determines where tab stops are placed. The number of spaces per tab is entered here, ranging from 1 — 9.

Expression Macros

Expression macros may be set to any text representing an expression. While debugging, a function may be pressed to evaluate the associated expression in the expression window. This feature is useful for variables which are frequently printed.

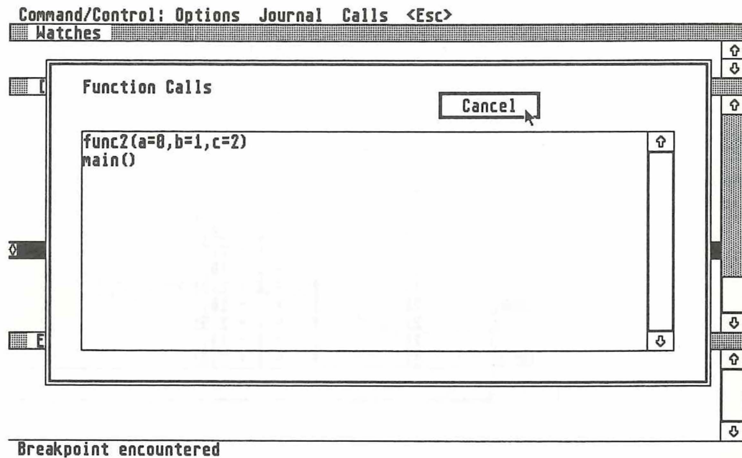
Journal

A journal of the last 100 source lines or machine instructions executed is kept by the debugger. The **Journal** command displays a list of these lines or instructions. The journal can be scrolled via the scroll bar and is dismissed with the "Cancel" button or the <Return> key. The keyboard shortcut <Ctl>-J also shows the journal dialog.



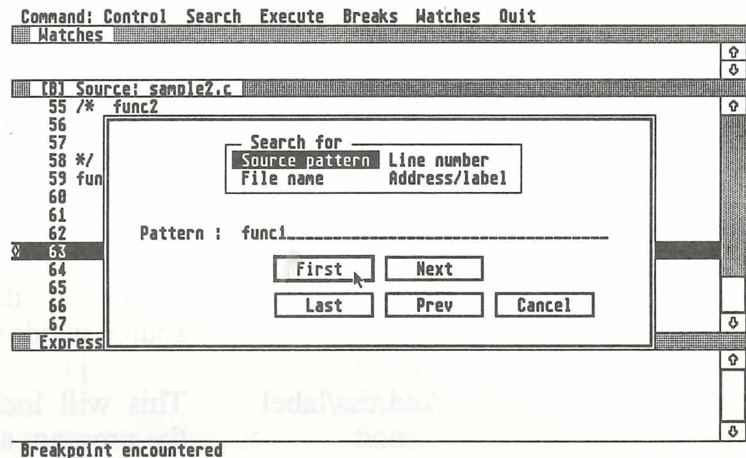
Calls

The **Calls** command displays a dialog showing the current calling order of functions and their parameters. The function calls are listed from the most recent call. The list can be scrolled and is dismissed with the "Cancel" button. In the assembly mode, the actual "JSR" or "BSR" instruction is disassembled.



Search

The Search command is used to find a string in the source window, the beginning of a particular source file, an address or label in the target executable, or any line number in the source window.



Search for

The collection of radio buttons at the top of the dialog determines the type of search to perform. The desired search type is selected with the mouse.

Source pattern — This search finds a pattern in the source code. If a match is found, the appropriate line is scrolled to the top line of the

	source window. In the assembly mode, the address which most nearly matches the located source pattern is scrolled to the top line of the window. The pattern rules are described below.
File name	This performs a pattern search similar to the "String" search, except that the pattern is only compared to source file names. If a file name matches the given pattern, the source file is scrolled to the top line in the source window.
Address/label	This will locate an address in the program and scroll the closest match to the top line. If a number is entered, it is assumed to be an address. Numbers may be entered in any format; hexadecimal, decimal, or octal, by using appropriate C literal syntax. For example, 0x2034 will scroll to that address in the source window.

	If a name is entered, it is assumed to be a GEM symbol name. The name is converted into an address and the appropriate line is scrolled to the top. For this type of search to work, the executable must contain GEM symbols, and assembly information must be loaded at startup.
Line number	Finds a line number in the source window. If found the window will be repositioned with the line number at the top of the window. In the assembly mode, line numbers are those of disassembled machine instructions.

Note

When searching for GEM labels, include the leading underscore or tilde. Labels in the TEXT, DATA, and BSS segments may be located.

Occurrence

The search is started by selecting one of the occurrence buttons or canceled with the “Cancel” button. The “Next”, “Previous”, and “Last” occurrences are only available when searching for a pattern in the source:

First	Finds the first occurrence of the pattern, or the only occurrence of a file name, address, or line number.
Last	Finds the last occurrence of the pattern in the source. The search is started at the end of the source and proceeds backwards.
Next	Finds the next occurrence of the pattern in the source from the last successful search. The shortcut <Ctl>-N can be used from the keyboard to do a search next.
Previous	Finds the previous occurrence of the pattern in the source from the last successful search. The shortcut <Ctl>-P can be used from the keyboard to do a search previous.

Search Patterns

A form of regular expression is used as a pattern when searching for a string or file name. Patterns are formed using the following rules:

^	Matches the beginning of a line.
$	Matches the end of a line.
*	Matches zero or more of any character.
?	Matches any single character.
\	A '\' followed by a single character matches that character. This is useful to match '$' and other special pattern characters.
[...]	A set is string enclosed by brackets and matches any single character in the set.

Any other character matches that character.

Example Searches

Here are some example searches using "Source pattern" as the search method:

```
^main
```

Searches for the beginning of a line followed by “main”.

```
array?
```

Searches for the string “array” followed by any single character.

```
printf*hello
```

Will search for “printf” followed by zero or more of any character(s), followed by “hello”.

Note

As per the C language, all name searches are case sensitive.

Breakpoints and Expressions

Breakpoints are special marks in the target program which will stop it's execution when encountered. They allow the target to be halted so that variables can be examined or changed. Breakpoints are also useful for allowing the program to run quickly to a certain function or line for subsequent single-stepping or tracing. For example, a loop which iterates 1000 times would be difficult to single step through. Setting a breakpoint on the line just after the loop and using the **Go** command will execute the loop at full speed and then stop.

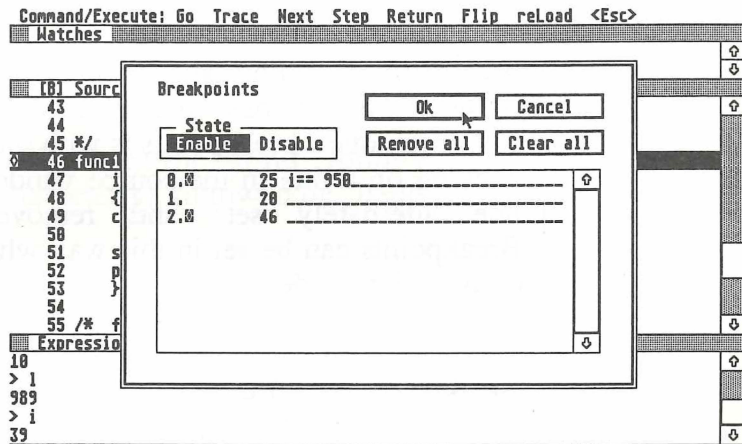
Breakpoint indicators appear in the source window and show which lines have breakpoints. The easiest way to add or remove a breakpoints is with the mouse, by simply clicking on a line in the source window. Clicking on a line alternately sets and removes a breakpoint. Breakpoints can be set in this way whether in the source or assembly mode.

Breakpoints can also be toggled on the current line using the '*' on the keypad, or they can be toggled on the top line of the window with the '/' key on the keypad.

The **Breaks** command displays a dialog which lists all breakpoints and allows for an optional "break expression" to be added to any breakpoint. Breakpoints here are indicated by a source line number if in the source

mode, or by the address of the breakpoint in memory if in the assembly mode. Breakpoints may not be added with this dialog, but they may be removed and their expressions may be edited.

The optional break expression, located in the text entry line after each breakpoint, is evaluated each time the breakpoint is encountered. If the expression evaluates to



non-zero, the program is halted, otherwise the break is not taken and program execution continues. Breakpoints which do not have expressions are always taken.

The Breakpoint dialog can be edited freely using the mouse or the <Tab> or arrow keys. The radio buttons labeled "State" can be used to enable or disable all breakpoints in the dialog. The source window title contains a "[B]" when breakpoints are enabled, or a "[-]" when breakpoints are disabled. The "Clear all" button erases all edit lines in the dialog and the "Remove all" button removes the break indicator from all breakpoints, thus marking them for removal. Individual breakpoints can be removed or restored by clicking on the break line number or address to the left of each line. When the dialog is closed, all breakpoints which contain the breakpoint indicator will remain, the rest will be removed. The "Ok" button will install the breakpoints and expressions, while the "Cancel" button will discard any changes.

Pass Counts

A pass count can be placed on a breakpoint by using a predeclared temporary. It works like this:

Set a breakpoint and bring up the "Breaks" dialog.

Enter in the expression line next to the desired breakpoint the pass expression, such as

```
$0++ == 9
```

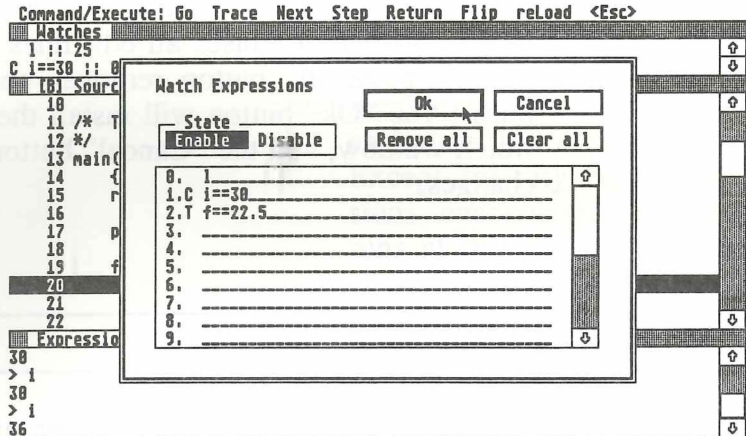
Close the dialog with “Ok” and make sure “$0” is cleared to zero by evaluating the following expression in the expression window:

```
$0 = 0
```

Now execute in the desired manner. When “$0” reaches 9, the break will be taken. The expression is only evaluated when the break is encountered.

Watches

Watches are expressions which are evaluated and printed each time the debugger gets control back from the target program — that is, after a Go, Trace, Next, Step, or Return (see page 44). Watches allow variables to be monitored, as the target runs. Watch expressions can also have a "watch termination" condition which will stop tracing and alert the user when the condition is met.



Conditions are set by clicking on the line number in the watch dialog scrolling window. Clicking will rotate through the three possible conditions; 'T', 'C', or no condition. If the 'T' condition is set, the user will be alerted with the message "Watch termination", and any tracing

will stop. If the associated expression is true (non-zero). If the 'C' condition is set, the user will be alerted each time the result of the expression changes. If no condition is specified, the result is printed but no message or interruption of tracing will occur.

The Watch dialog operates identically to the Breakpoint dialog. The radio buttons labeled "State" can be used to enable or disable all breakpoints in the dialog. The "Clear all" button erases all edit lines in the dialog and the "Remove all" button removes conditions from all lines. The "Ok" button will install the watches into the watch window, and the "Cancel" button will discard any changes.

Note

Remember that Watch expressions are only evaluated when the debugger screen is entered after coming back from the target. This means that conditional watches cannot be used to stop a program during a **Go** command. To conditionally stop a program in this way, see "break conditions."

Execution

LaserDB provides a variety of ways to control target execution. The current mode affects the operation of the execution commands. Any mode of execution will be halted by a breakpoint or by a processor exception. Execution cannot continue after a processor exception — the target must be reloaded.

Go

The **Go** execution command runs the target unhindered by watch expressions. Breakpoints are active if enabled and break expressions are evaluated each time they are encountered. Except for any break expressions encountered, the target runs at full speed. Execution can only be stopped by a breakpoint, a processor exception, or with the <Alternate>-<Help> key combination. Note that <Alternate> -<Help> will only work if the PC is *not in ROM*.

Trace

Trace repeatedly executes the Next or Step command, evaluating watch expressions after each line is executed (see below). A small dialog allows either Next or Step to be cho-

sen. Tracing can stopped by pressing any key or by clicking either mouse button. Watch termination will also stop tracing as described on page 59.

Next

Next executes until the next source line. Function calls are-executed as one source line and are not stepped into.

Step

Similar to the Next command, **Step**, executes until the next source line. The difference, however, is that function calls are stepped into rather than over.

Return

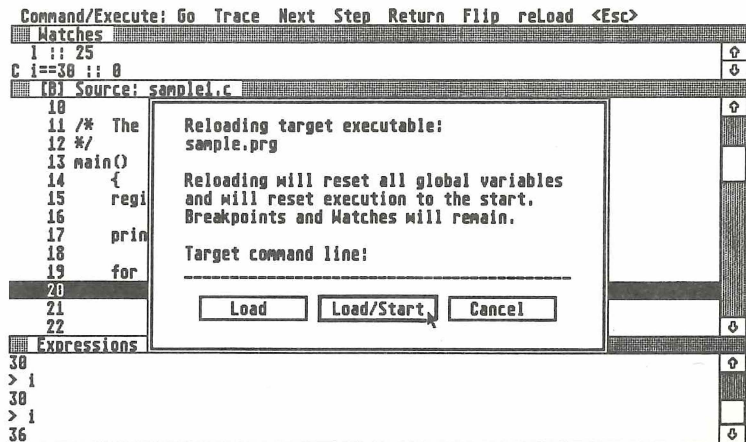
Return will execute until the current function returns to it's caller. It is useful when a function is accidentally stepped into.

Flip

The **Flip** command displays the target's screen. Since the target is not executing, there may be no interaction with it. Pressing any key or either mouse button returns to the debugger screen.

reLoad

reLoad reads the executable target from disk back into memory, resets all uninitialized program globals back to zero, and resets the current line to the first instruction of the program. reLoad has no effect on break points, watches, or any other debugger settings. Note that the 'L' key is used to select this command from the keyboard since the 'R' is used by Return.



Quitting

The **Quit** command exits the debugger. The configuration file is automatically saved at this time, recording the current window positions and the user settings from the Options dialog as well as the settings from the Startup dialog.

CHAPTER 6
SAMPLE SESSION

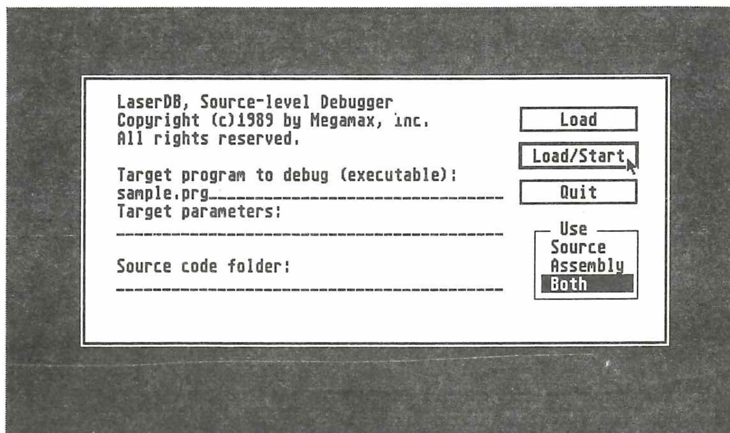
Try this sample session.

Step 1

Make a copy of the LaserDB distribution disk and put the original away.

Step 2

Insert the copy just made and from the GEM desktop, double-click on LDB.PRG.



Step 3

When the Startup dialog appears, just press <Return> or click on “Load/Start”. The configuration file has already been set for this example. The program being debugged is SAMPLE.PRG. It was compiled from two source files called SAMPLE1.C and SAMPLE2.C. When “Load/Start” is pressed, the executable and the source will be loaded, and a **Step** will be executed which will step from the initialization code *into* the function “main()”.

```
Command: Control  Search  Execute  Breaks  Watches  Quit
 Watches
 [B] Source: sample1.c
   10
   11 /*  The function main()
   12 */
   13 main()
   14     {
   15     register int        i;
   16
   17     printf("This is a sample\n");
   18
   19     for ( i = 0; i < 1000; i++ ) {
   20         l   = i - 10;
   21         f   = l + 2.5;
   22         }
 Expressions - <Tab> to enter
```

Step 4

The screen should appear as above. The function "main()" is ready to be entered.

Choose **Execute** in the menu bar. 'E' from the keyboard or a mouse click on the word "Execute" will do this. These are the execution commands.

Now choose **Next**. The current line will move to the "printf()" line.

Choose **Next** again to execute through the "printf()" function call.

Choose **Flip** to see the target's screen. Any key or a mouse click will return to the LaserDB screen.

Choose **Next**. The "for" loop has now begun.

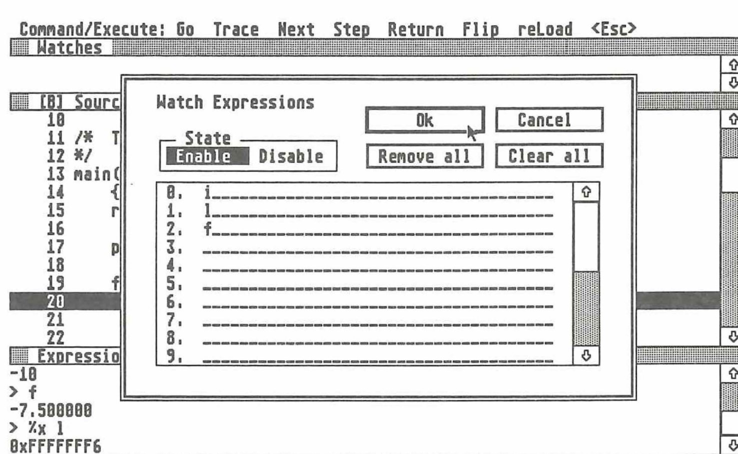
```
Command/Execute: Go  Trace  Next  Step  Return  Flip  reLoad  <Esc>
 Watches

 [0] Source: sample1.c
  10
  11 /*  The function main()
  12 */
  13 main()
  14     {
  15     register int        i;
  16
  17     printf("This is a sample\n");
  18
  19     for ( i = 0; i < 1000; i++ ) {
  20         l   = i - 10;
  21         f   = l + 2.5;
  22         }
 Expressions - <Tab> to enter
```

Step 5

Press <Tab> to evaluate an expression. Enter "i" and press <Return>. This will print the value of the local variable "i". It is now zero. Try again, but this time try to print the global "l". And again to print the global "f".

Choose **Next**, and then **Next** again. Execution is now looping. Try printing the variables again and executing again. The will changed according to the assignments in the code.

Try printing with a format using "%x l". This will print "l" in hexadecimal.



Step 6

Open the **Watch** dialog. Go back to the main menu with the **<Esc>** item, or type <Ctl>-W directly from the **Execute** menu. Enter the three variables as shown above. Make sure the "Enable" item is selected. Click "Ok" when done.

Choose **Trace** and and watch it go. Any key or a mouse click will stop the trace.

When your through experimenting here, bring back up the **Watch** dialog and select "Clear all" to erase all watches.

```
Command/Execute: Go  Trace  Next  Step  Return  Flip  reLoad  <Esc>
 Watches

 [B] Source: sample1.c
    17      printf("This is a sample\n");
    18
    19      for ( i = 0; i < 1000; i++ ) {
    20          l   = i - 10;
    21          f   = l + 2.5;
    22          }
    23
    24      for ( i = 0; i < 100; i++ ) {
    25          func1(i);
    26          func2(i,i+1,i+2);
    27          }
    28
 Expressions - <Tab> to enter
-10
> f
-7.500000
> %x l
0xFFFFFFF6
```

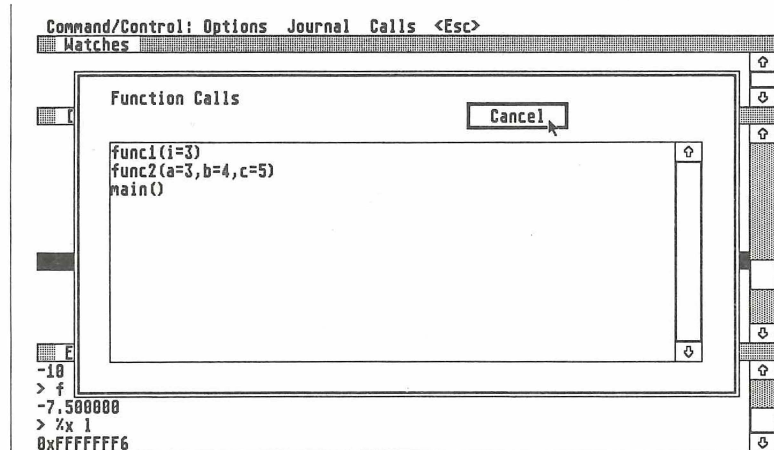
Step 7

Set a breakpoint on the line shown above (on line 24). Do this by clicking with the mouse anywhere on the desired line. Notice the indicator in the left most column.

Now choose **Go**. This will execute at full speed the rest of the way through the loop and stop at the break line.

Try pressing the '*' key on the keypad. Notice that this is the same as clicking on the current line with the mouse.

Choose **Next** to start the loop, and then choose **Step** to enter the function call. Try using **Next** and **Step** to get a feel for how they work.



Step 9

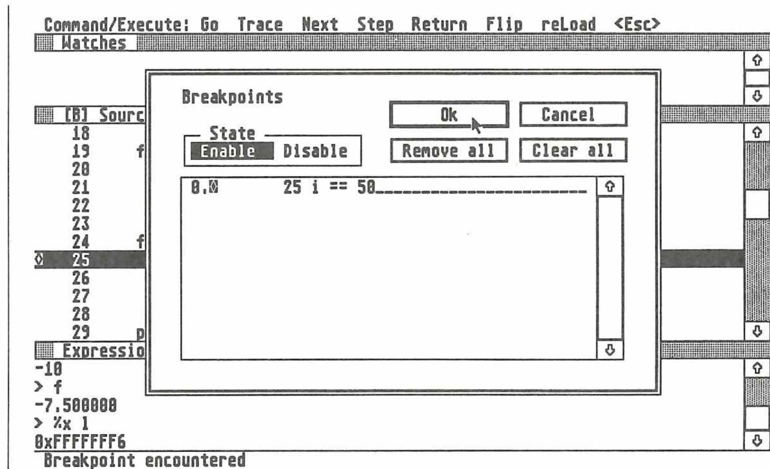
Choose **Step** until the current line is on the "sprintf()" in the function "func1()". Then choose **Calls**, also from the **Control** menu. This shows the current function calling order and the parameters of each. Try this when "main()" calls "func2()" which calls "func1()."

Now choose **Return** and try **Calls** again.

```
Command/Execute: Go  Trace  Next  Step  Return  Flip  reLoad  <Esc>
Watches

Execution Journal
                                   Cancel
   53     }
   63     func1(a);
   46 func1(i)
   51     sprintf( buffer, "%d\n", i );
   52     puts( buffer );
   53     }
   64     func1(a);
   46 func1(i)
   51     sprintf( buffer, "%d\n", i );
   52     puts( buffer );
   53     }
   65     }
-10
> f
-7.500000
> %x i
0xFFFFFFF6
```

Step 8

Choose **Journal** from the **Control** menu. <Ctl>-J is the shortcut for this command. This shows all source lines traced or stepped. In the assembly mode, disassembled instructions are shown here.



Step 10

Set a breakpoint on the line shown above and choose **Go** or **Next** until the breakpoint is encountered. Now open the **Breaks** dialog by choosing it from the main menu, or by typing <Ctl>-B. The breakpoint on line 25 is the one just set. Enter the break expression shown above. Close the dialog with "Ok". Make sure "i" is less then 50. If it is greater than 50, enter the following expression using the <Tab> command:

```
i = 0
```

Choose **Go** to start execution. The loop will run, evaluating the expression only when the breakpoint is hit. The break will only be taken when the expression is true (non-zero).

```
Command/Execute: Go  Trace  Next  Step  Return  Flip  reLoad  <Esc>
 Watches                                                      Regs.
                                                              D0 0000000A
                                                              D1 00000003
                                                              D2 00000000
 [B] Assembly: TEXT Segment                                   D3 00000000
 00042D62        BLT.S      0x00042D22                        D4 00000000
 00042D64        CLR.W      D7                                D5 00000000
 00042D66        BRA.W      0x00042D8E                        D6 00000000
 00042D6A        MOVE.W     D7,-(A7)                          D7 00000032
 00042D6C        JSR        _func1.L                          A0 000475DA
 00042D72        ADDQ.L     #2,A7                             A1 00049738
 00042D74        MOVE.W     D7,D0                             A2 00000000
 00042D76        ADDQ.W     #2,D0                             A3 00000000
 00042D78        MOVE.W     D0,-(A7)                          A4 000456F6
 00042D7A        MOVE.W     D7,D0                             A5 00042B4A
 00042D7C        ADDQ.W     #1,D0                             A6 00047866
 00042D7E        MOVE.W     D0,-(A7)                          US 00047862
 Expressions - <Tab> to enter                                 SS 00007554
-7.500000
> %x 1                                                        PC 00042D6A
0xFFFFFFF6                                                    CCR xNzvC
> i=0                                                             User
0                                                             Mask: 3
Breakpoint encountered
```

Step 11

Using the Options dialog, switch to "Mode-Assembly" mode and select "Display-Registers." All debugger functions as described above work consistently in the assembly mode. <Ctl>-M can be used to quickly change the mode. Clicking on the register window toggles between the stack display and the register display. The small arrow shows the current frame pointer (register A6). Execution and breakpoints use machine instructions rather than source lines, but otherwise work identically.

Registers can be used in expressions. For example try evaluating:

```
%d $D0
```

in the expression window. It will print the value of register "D0" in decimal format.

Step 11

When your through with this sample session, choose **Quit** to return to the desktop. The configuration file will be written out at this time so that all settings will be remembered between sessions.

APPENDIX
KEYBOARD SHORTCUTS

Key	Meaning
‘ +’/‘-’ (keypad)	Scroll the expression window up/down.
‘*’ (keypad)	Set or remove a breakpoint at the current line.
‘/’ (keypad)	Set or remove a breakpoint at the top line of the source window.
<Ctl>-B	Bring up the Breakpoints dialog.
<Ctl>-C	Diaplay the function calls dialog.

<Ctl>-J	Bring up the Journal dialog.
<Ctl>-L	Toggle line numbers in the source window.
<Ctl>-M	Toggle the mode between source and assembly. This is available only "Both" was selected at startup.
<Ctl>-N	Search for the next occurrence of a source pattern.
<Ctl>-O	Open the Options dialog.
<Ctl>-P	Search for previous occurrence of a source pattern.
<Ctl>-R	Toggle the Register window on and off.
<Ctl>-S	Bring up the search dialog.
<Ctl>-W	Bring up the Watch dialog.
<Enter> (keypad)	Reposition the source window to show the current statement.
<Space bar>	Single step execution of the target — the Step command.

Help	Get keyboard shortcut help.
Home (shift)	Reposition the source window to the end.
Home	Reposition the source window to the beginning.
Return	Issue the Next command.
Up/down arrow (shift)	Scroll the source window up/down by one line.
Up/down arrow	Scroll the source window up/down by one line.

Index

"Assembly" button. 14
"Both" button. 14
"Go" execution. 61
"Load" button. 14
"Load/Start" button. 15, 16
"Next" execution. 62
"Quit" button. 15
"Return" execution. 62
"Source" button. 13
"Step" execution. 62
"Trace" execution. 61
-V flag. 3
-Z flag. 9, 11

A

Assembly. 6
Assembly mode. 37

B

Breakpoint indicators. 55
Breakpoint pass count. 57
Breakpoints. 39, 55

C

C operators. 31
Calls dialog. 48
CC.TTP. 9
CCOM.TTP. 9
Choose. 5
Click. 5
Compilation. 9
Current line. 18, 39
Current scope. 18
Cursor. 5

D

Debugger information. 8
Dialog boxes. 44
Drag. 5

E

Expressions. 17, 41

G

GEM symbols. 19

I

Identifiers. 19
Insertion point. 5

J

Journal dialog. 47

L

LaserDB. 1
LD.TTP. 11
LDB.CFG. 12
LDB.PRG. 11
Lexical scope. 19
Line number operator. 25
Linking. 10
Literals. 22

M

Memory checksum. 24
Menu bar. 34

O

Options. 45

P

Predeclared temporaries. 27
Predeclared variables. 26
Printing formats. 28, 29
Processor registers. 27

Q

Quit command. 64

R

Register window. 43
Reload dialog. 63
Resizing windows. 35

S

Scrolling windows. 35
Search dialog. 49
Search patterns. 53
Select. 6
Source mode. 37
Startup. 11

T

Text entry. 7
Type casting. 23

W

Watch conditions. 59
Watch dialog. 60
Watches. 59

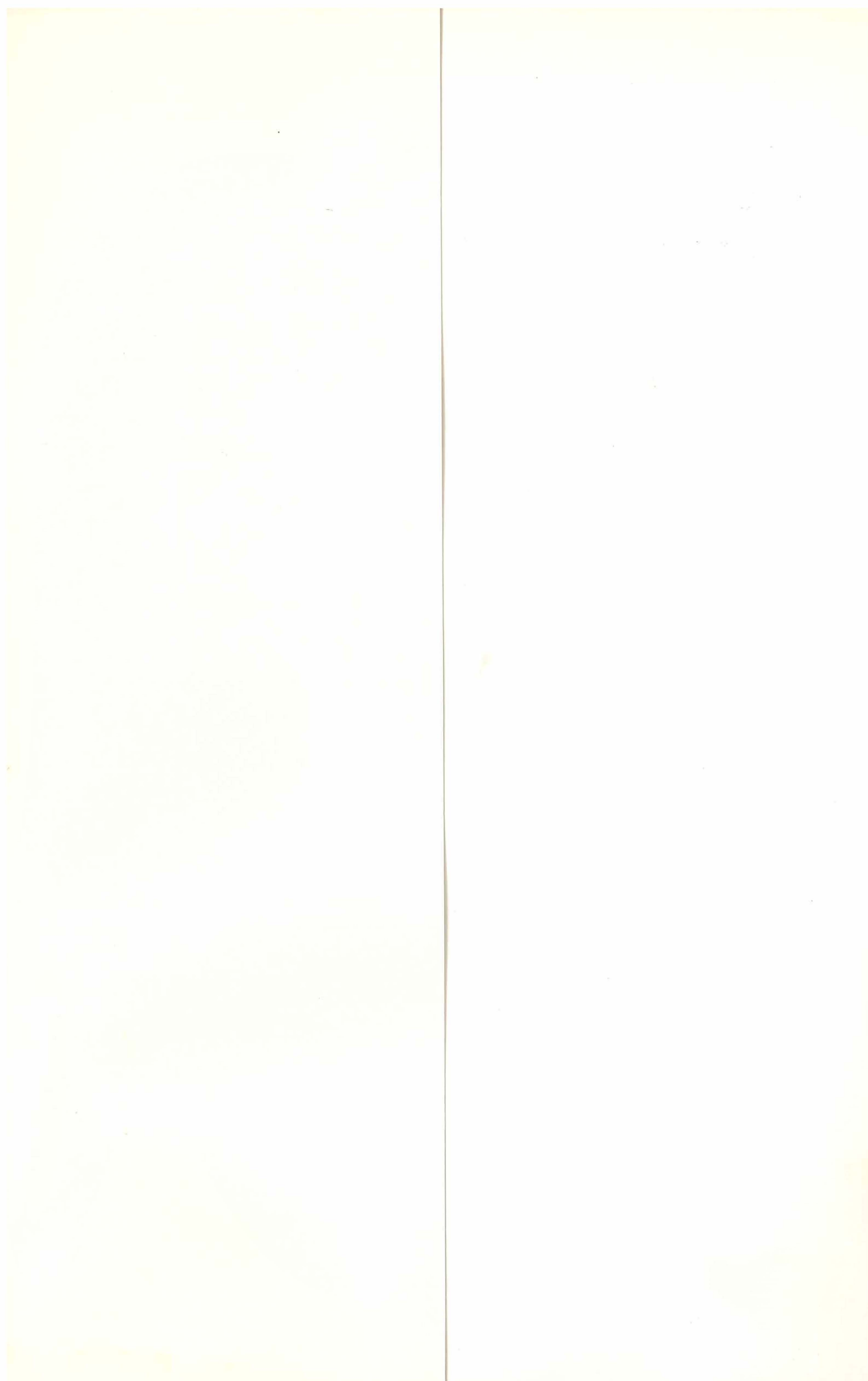
Notes

Notes

Notes

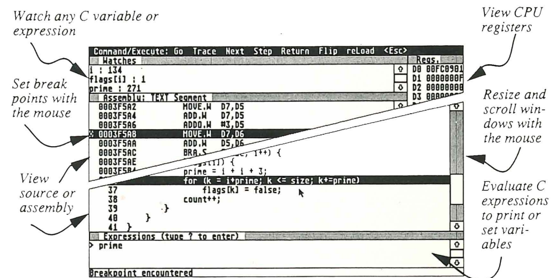
Notes

Notes



```
Command/Execute: Go  Trace  Next  Step  Return  Flip  reLoad  <Esc>
 Watches
i : 134
flags[i] : 1
prime : 271
 Source: sieve.c
    28  Supexec(gettime);
    29  for (iter = 1; iter <= 10; iter++) {
    30      count = 0;
    31      for (i = 0; i <= size; i++)
    32          flags[i] = true;
    33      for (i = 0; i <= size; i++) {
    34          if (flags[i]) {
    35              prime = i + i + 3;
◊   36              for (k = i+prime; k <= size; k+=prime)
    37                  flags[k] = false;
    38              count++;
    39          }
    40      }
    41  }
 Expressions (type ? to enter)
> prime

Breakpoint encountered
```

```
Command/Execute: Go  Trace  Next  Step  Return  Flip  reLoad  <Esc>
 Watches                                                  Regs.
i : 134                                                   D0 00FC0901
flags[i] : 1                                              D1 0000000F
prime : 271                                               D2 00000000
 Assembly: TEXT Segment                                   D3 00000000
  0003F5A2        MOVE.W   D7,D5                          D4 00000000
  0003F5A4        ADD.W    D7,D5                          D5 00000003
  0003F5A6        ADDQ.W   #3,D5                          D6 00000000
◊ 0003F5A8        MOVE.W   D7,D6                          D7 00000000
  0003F5AA        ADD.W    D5,D6                          A0 00041FCC
  0003F5AC        BRA.S    0x000E                         A1 00047F56
  0003F5AE        LEA.L    0x00041FCC.L,A0                A2 00000000
  0003F5B4        CLR.B    0(A0,D6.W)                     A3 00000000
  0003F5B8        MOVE.W   D5,D0                          A4 00041F08
  0003F5BA        ADD.W    D0,D6                          A5 0003F370
  0003F5BC        CMPI.W   #0x1FFE,D6                     A6 00046076
  0003F5C0        BLE.S    0x00EC                         US 0004605C
  0003F5C2        ADDQ.W   #1,D4                          SS 00007554
  0003F5C4        ADDQ.W   #1,D7
 Expressions (type ? to enter)                            PC 0003F5A8
> prime                                                   CCR xnzvc
                                                             User
                                                          Mask: 3
Breakpoint encountered
```



LaserDB

Source-level and Assembly-level Debugging!

- Powerful and easy to use dual mode debugger — view source code or assembly.
- Debug GEM programs and games — even debugs low-resolution programs.
- Mouse based user interface — resizeable windows, scroll bars, button menu bar, dialog boxes.
- Simple, logical command structure with numerous keyboard shortcuts.
- Print or set variables with C expressions. Access global, local, and register variables. Special syntax extensions to access processor registers.
- Set and remove breakpoints with the mouse! Optional break expression allows conditional breaks.
- Watch window allows monitoring of C variables with option to break execution if a variable changes.

Requires Laser C to debug at the source level.
Laser C and LaserDB sold separately.

© Megamax, Inc., 1989

- Multiple execution modes:

Go	Execute until breakpoint or error.
Trace	Watch the source or disassembly while the program runs.
Next	Single step to the next source line or machine instruction without stepping into function calls.
Step	Single step to the next source line or machine instruction stepping into function calls.
Return	Execute until the current function returns.

- Execution journal shows a history of source lines or machine instructions in the order they were executed.
- View function calls in the order of their activation on the stack.
- Debug any GEM executable [illegible] bly mode.