

A LISP DEBUGGER FOR DISPLAY TERMINALS

by

Daniel Conrad Halbert

Submitted in Partial Fulfillment

of the Requirements for the

Degree of Bachelor of Science

at the

Massachusetts Institute of Technology

May, 1978

Signature of Author _____
Department of Electrical Engineering and Computer Science, May 22, 1978

Certified by _____
Thesis Supervisor

Accepted by _____
Chairman, Departmental Committee on Theses

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

Copyright © 1978 by Daniel C. Halbert

A LISP DEBUGGER FOR DISPLAY TERMINALS

by

Daniel Conrad Halbert

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 1978 in partial fulfillment of the requirements for the Degree of Bachelor of Science

Abstract

DIDL is a LISP debugger that makes extensive use of the capabilities of modern display terminals. The DIDL debugger has two main objectives: to be a comprehensive debugging tool, and to use text display terminals in the flexible way in which they should be used, thinking of their screens as selectively modifiable pictures of text.

DIDL's debugging capabilities include display of the state of an interrupted program, the ability to examine and modify the data environments created by the program, conditional breakpoints, single-stepping of code, and dynamic checking for any condition while a program is running. The display features of DIDL emphasize showing user program code in context: for instance, when the user program hits a breakpoint, DIDL automatically shows not only the point at which the breakpoint occurred, but also the surrounding code. Single-stepping moves a cursor through the user code, showing exactly what is to be executed, in context. The user can move around in his or her code as easily as using a real-time display editor, setting breakpoints and inserting patches. Extraneous, transient information is removed from the screen, while more important material remains.

Thesis supervisor: Peter Szolovits, Assistant Professor of Electrical Engineering and Computer Science

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

Copyright © 1978 by Daniel C. Halbert

Acknowledgements

I would like to express my appreciation and thanks to:

Peter Szolovits, for suggesting a fine thesis topic which taught me a great deal and opened new areas of interest, for very valuable discussions, and for pulling me out of ruts with suggestions, solutions, and words of encouragement;

Alan Bawden, Dan Weinreb, and Dave Moon, for patiently answering my endless questions about MACLISP, and for LISP macros, which made my coding so much easier;

Richard Stallman, for EMACS, without which my editing would have taken five times as long, and from which I used many valuable ideas;

Howard Cannon, for knowing MACLISP inside-out, and helping me whenever I asked;

Ted Anderson, for inventing the concept of self-referencing acronyms;

my fellow members of the Student Information Processing Board, for good ideas, good discussions, and good food, no matter what the hour (coffeehouse, anyone?);

and finally to HIC, KMP, RWK, BKERNS, DRB, BAK, and CARL, for being terrific company at two in the morning, and later, and later...

CONTENTS

1. Introduction	7
2. Debuggers	8
2.1 Machine language debugging	8
2.2 Source language debuggers	8
2.3 Capabilities of debuggers	10
3. Display terminals	12
3.1 History	12
3.2 Virtues	12
3.3 Disadvantages	13
3.4 Uses	13
4. Debuggers with display capabilities	15
4.1 Machine language debuggers for displays	15
4.2 MEND	15
4.3 Teitelman's Display-Oriented Programmer's Assistant	15
4.4 COPILOT	16
4.5 Conclusions	17
5. Basic Capabilities of DIDL	18
5.1 DIDL displays	18
5.2 DIDL's debugging capabilities	21
6. How DIDL works	24
6.1 Displaying user code	24
6.2 Stack frame examination and manipulation	26
6.3 Breakpoints and stepping	27

7. Implementation notes	30
7.1 The single process problem	30
7.2 Coding	31
8. Future work	32
8.1 Dynamic watching	32
8.2 Debugging data as well as code	32
8.3 Complete interactive programming systems	32
8.4 Human engineering	33
9. Conclusions	34
9.1 Context displays	34
9.2 Understanding displayed text	34
Appendix I. DIDL Commands	35
Appendix II. References	37

FIGURES

Fig. 1. A DIDL User Function Display	19
Fig. 2. A DIDL Stack Frame Trace Display	20

1. Introduction

DIDL is a LISP debugger that makes extensive use of the capabilities of modern display terminals. DIDL is for use with the MACLISP [Moon] dialect of LISP, and is written in that language, using the built-in primitive features of MACLISP that already exist for debugging.

The DIDL* debugger has two main objectives:

- to be a comprehensive debugging tool, using the debugging ideas found in existing MACLISP debuggers and adding features not currently available, and
- to use text display terminals in the flexible way in which they should be used, thinking of their screens as selectively modifiable pictures of text.

DIDL's debugging capabilities include display of the state of an interrupted program, the ability to examine and modify the data environments created by the program, conditional breakpoints, single-stepping of code, and dynamic checking for any condition while a program is running. The display features of DIDL emphasize showing user program code in context: for instance, when the user program hits a breakpoint, DIDL automatically shows not only the point at which the breakpoint occurred, but also the surrounding code. Single-stepping moves a cursor through the user code, showing exactly what is to be executed, in context. The user can move around in his or her code as easily as using a real-time display editor, setting breakpoints and inserting patches. Extraneous, transient information is removed from the screen, while more important material remains.

*DIDL is an acronym standing for Display Debugger for LISP, or Dan's Interactive Debugger for LISP, or DIDL Is a Debugger for LISP, or something else not yet thought of.

2. Debuggers

2.1 Machine language debugging

Programs which assist the user in debugging his or her programs have long been recognized to be enormous time-savers in the debugging process. In the very early days of computers, the programmers of a machine were also its operators, and so they frequently had the opportunity to debug their programs directly from the machine's console. They single-stepped the program, examined the memory state at various points in the code, and patched code to try fixes. But when the "inefficiency" of such techniques was realized (since one couldn't afford to tie up such expensive machines for so long), debugging was thereafter done away from the machine, using such primitive aids as memory dumps. Core dumpers were the first debugging programs, and they gradually became more and more sophisticated.

But timesharing systems, and (relatively) inexpensive minicomputers were appearing. Programs were written that would simulate console debugging, in a much more convenient and powerful fashion. Now one didn't worry about tying up a machine, either because other users could simultaneously share in the use of the machine, or because the cost of the computer was low enough not to worry about wasting computing resources during idle machine time. So the first interactive debuggers appeared (see [Evans] for some history). These tools were useful not only for debugging machine language programs, but could also be used for debugging compiled higher-level languages if the user understood the structure of the compiled code, and if the compiler provided some information, such as a storage map showing the locations of statements and variables.

2.2 Source language debuggers

Though machine language debuggers are very useful for debugging machine and assembly language problems, they provide too much information at the machine-language level for easy debugging of programs written in higher-level languages (HLL's). The user would like to see the abstract machines represented by the HLL's, and not the underlying real machine. In fact, most machine language debuggers actually pretend to debug a machine that can understand symbolic names for instructions and memory addresses; that is, they emulate the assembly language model of a machine. This is a low-level but important abstraction because it does mirror the language the assembly language programmer is writing in.

Without interactive debuggers, HLL program debugging is typically done by inserting extra statements in the program to monitor its behavior. For instance,

A LISP Debugger for Display Terminals

statements to print out the values of variables which one suspects of having gone awry would be inserted wherever the variables changed. Such tracing features were frequently incorporated as extensions of an HLL; the ubiquitous TRACE facility found in many FORTRAN compilers (cf. [IBM]) is an example of such a debugging aid.

With such aids, the programmer can get an idea of the dynamic flow of a program, instead of relying on a static picture such as a memory dump. However, there is still no way to stop the program before or while it is doing something wrong, and then debug it interactively.

Interactive debugging for HLL's was not common for many years because HLL programs were typically run in a non-interactive batch environment. Gradually, as timesharing systems became more common, the desire for interactive HLL debugging arose. For some interpreted interactive languages, it is unnecessary to provide extra debugging facilities. For example, APL and LISP allow the programmer to examine and change variable bindings at will after interrupting program execution. With other interactive languages such as BASIC, it is easy to insert tracing-type statements into the program. For most interpreted languages, it is usually relatively easy to add extra features to aid in debugging, because a large amount of useful information is kept around by the interpreter for use in running the user's program. For instance, the symbol table and source code are almost always available. And since the interpreter is running the program (or more appropriately, emulating a machine in which the HLL runs), it can monitor the running of the program closely.

Creating a debugger for compiled HLL's is not nearly as easy, because much information (such as variable names and memory addresses, and where statements begin and end) is lost in the translation process. So the compiler must provide such information for use by the the debugger, outputting it in addition to the object code. Since the object code does not run under the close supervision of another program, the debugger must exert control over the execution of the object code when necessary, and must understand its format and idiosyncrasies. The compiler can insert extra code in the object code to aid the debugger, but this may add substantially to the size and execution time of the user's program. Essentially, debugging compiled HLL code at the source language level requires careful interfacing between the compiler and the debugger. If either program needs to change substantially, or wants to expand its capabilities, the other program and the interface mechanism may have to change as well.

Despite these obstacles, many debuggers for compiled HLL's exist. A few examples: BDDT [BBN BCPL] is a debugger for the systems programming language BCPL; probe [Honeywell] is a debugger on the Multics system capable of working

A LISP Debugger for Display Terminals

with both FORTRAN and PL/I; BAIL [Reiser] is a debugger for SAIL, an ALGOL-like language. The author is familiar with BDDT and probe; they both save substantial amounts of time compared with the machine language debuggers that were used before they existed.

2.3 Capabilities of debuggers

Most debuggers provide some subset of the operations available from the console of a computer. However, the "machine" on which one is debugging may not be the actual machine the program runs on. Instead, the debugger for a particular language can make it seem as if the user's program is running on a machine which directly interprets that language. For a machine, virtual or real, the debugger usually provides the following capabilities:

1. Interrupt the running program.
2. Examine and change the contents of memory locations (change and modify variable bindings for HLL's).
3. Set breakpoints.
4. Patch the program.
5. Run the program one statement at a time (single-stepping).

Despite its universality, this "front-panel" debugging paradigm can be improved. The functions above are all accomplished by stopping the program to modify it or to look at its data environment. There are no capabilities for examining the changing state of a running program. Such features are frequently not put into hardware because they are difficult or expensive to implement. But one would like to be able to stop a running program when:

1. The memory becomes a particular state.
2. A particular memory location (or variable) is accessed.

These extra features are essentially "break on machine state" and "break on data access". The regular kind of breakpoint is a "break on instruction access", but is usually implemented not by detecting the access, but by substituting a special break instruction to be executed instead of the instruction at the place we wish to break. In essence, we have tagged the instruction to be broken on. If we could tag data, breakpoints on data access would be just as easy. Unfortunately, few machines provides this capability.

Detecting a particular memory state is more difficult, because the condition on which to break may be very complicated. It may be infeasible to install such a feature

as a basic machine capability, because it would require possibly looking at many memory locations. Debuggers implement these dynamic checks by simulating or single-stepping the program; in other words, by taking complete control of its execution.

Most debuggers have some or all of these features. In addition, debuggers, especially HLL debuggers, usually have some data and program environment display features to show information that would otherwise be troublesome or impossible for the user to get. For instance, a language may use a stack mechanism for procedure calling, and so a debugger stack frame display would be useful to the programmer. In this case, the stack mechanism is an integral part of the abstract machine created by the HLL, but the HLL itself has no way to access the contents of the stack directly. So the instructions of an HLL machine could be insufficient to allow access to all useful debugging information. In such cases, the HLL debugger must enlarge the definition of the HLL machine.

3. Display terminals

We define a display terminal to be a computer terminal that displays an image, and is capable of modifying or replacing any part of that image. Thus whatever the terminal presents to the user is not permanent, as in the case of printing terminals, but inherently transient. The most common display terminals today are cathode-ray tubes, which have oscilloscope and television technology in their ancestry. Most display terminals are very fast compared with printing terminals or plotter devices; their speed is their one of their primary virtues. We are mostly concerned with text display terminals.

3.1 History

Display terminals fall into two categories: those that can display arbitrary graphic images, and those that can display only text; the former correspond to hardcopy plotter devices, the latter to most printing computer terminals.

The first applications for display terminals were for plotting graphs and making line drawings. Displaying text was an incidental feature; it was frequently done by drawing each character as a number of lines. Terminals specifically designed for displaying text (and usually only text) came later, and were used primarily for business transaction systems which involved data entry or the viewing of a large amount of data in a short time. A typical application would be as a terminal in an airline reservation system, where speed is important.

Display terminals were relatively expensive, because unless they used a CRT storage tube, they required memory to hold the current screen information so that the display could be refreshed. Memory was expensive: a text display terminal might require 2000 8-bit words of memory, which, though not quite the size of a small computer's memory, was nevertheless a substantial fraction of it. However, in the past ten years or so, the price of memory and the rest of the electronics necessary for a display terminal has gone down sharply. Very recently, simple text display terminals have become cheaper than simple printing terminals, and so there is a tendency to replace printing terminals with display terminals.

3.2 Virtues

The two main virtues of a text display terminal are its speed and its selective update ability. The screen of a typical terminal can be filled in a few seconds. But the ability to change an already displayed image, or to preserve part of an image

while the rest changes is what makes display terminals unique.

Because the terminal is so fast, and also because displaying something consumes no obvious resources such as paper, screen terminal users are more inclined to display something when they feel it could help. For instance, they might idly peruse a text file looking for something even if they had a printed copy of the same thing nearby. The non-consumable nature of the display screen removes a psychological block of using up something when requesting a display. The only resource consumed is machine time, which is visible to the user only as a delay, if at all.

Display terminals also provide methods for pointing at objects on the screen. One can simply move the display cursor to a particular spot to highlight something, or if the terminal supports it, change a section of the display to the negative image of what was there (reverse video), or even to a different color.

3.3 Disadvantages

The chief disadvantage of a display is its ephemerality. Valuable output can easily be lost if too much output appears subsequently. And the display screen is of a limited size on which only a certain amount of output can appear.

Programs should usually be aware that their output is going to a display terminal. If their output is expensive to generate or impossible to recreate, it should be saved for possible redisplay. Programs should not indiscriminately send large amounts of output to the terminal, because the user may want to retain older, more valuable output. A program should use the selective update feature of displays to retain important output while reusing the display space of incidental output. Many good software support systems for display terminals implement the concept of "windows" on a display screen; there are two or more independent and non-conflicting areas on the screen which the program or the user selects for input and output.

3.4 Uses

Though display terminals are much more flexible than printing terminals, their capabilities are not usually extensively exploited. One reason is that they have only recently become as commonplace as printing terminals, and there is some time lag before the software that uses their unique features is written. Their recent replacement of some printing terminals also points up another problem: software must frequently be able to run using both kinds of terminals. Since the capabilities of standard printing terminals are mostly a subset of those of display terminals, leaving

old software as it is or writing new software that assumes the user is on a printing terminal saves work.

However, display terminals have been exploited for their unique properties in some cases. The most common display-oriented use of such a terminal is for text editing [Stallman] [Cicarelli] [BBN WE]. The primary advantage display terminals give in text editing is the ability to show a change immediately, in context. This ability is extremely important, and can be succinctly stated as "what you see is what you get".

The text editing paradigm used on printing terminals is to go to a place in the text that needs to be edited, then perhaps printing out several lines to confirm that one is editing in the right place. Then the editing commands are given, and some context is again printed to assure that the changes have been made correctly. Display editors, on the other hand, automatically provide this context. Some update the displayed text after a sequence of commands has been given by the user, while others (so-called "real-time" editors) update after every command. The latter also normally have single keystroke commands, so that a single action by the user results in a display change. This has important psychological benefits, because once the user learns the editor commands, the keyboard becomes an extension of his or her thoughts. The user become unaware of the interface between thought and action; he or she "points" to a place and then changes it.

The context provided by a display editor is extremely useful, and can substitute for the usual context providing mechanism, which is a listing of the text to be edited. It is frequently easier to scan through text using a display editor than to look through the paper copy. Indeed, the paper copy usually becomes superfluous except for reference use and perusal away from the terminal. The usual cycle of editing a program, running it, noting problems, re-editing it, and so on, can be done entirely from the terminal without reference to materials not displayable on the screen. The copy of the program viewable in the editor is always up-to-date, and changes do not have to be made twice: once in the actual program text, and once in the listing copy.

4. Debuggers with display capabilities

Debuggers that use display terminals effectively are even rarer than display text editors, possibly because they are less general tools than text editors, and because debuggers for use on serial stream output devices, such as printing terminals, are generally less frustrating to use than text editors for similar devices. Perhaps in the user's mind, debugging is inherently more complicated than editing, and so he or she is more patient and less inclined to frustration when debugging than when editing. Several display debuggers are discussed below.

4.1 Machine language debuggers for displays

There are several machine language debuggers that use display terminals effectively. The most well-known one is RAID [Stanford], a member of the DDT [Kotok] [Deutsch] [DEC] [MIT] family of debuggers. RAID's main feature is the automatic display of the contents of selected memory locations when the program is interrupted or while it is being stepped. Another debugging aid is ESP (Execution Simulator and Presenter) [Galley ESP] which simulates a running program while dynamically displaying data of interest. ESP, unlike RAID (or DDT), shows the context around an instruction being executed rather than just that one instruction. It also can draw lines to indicate branches and loops. Other simpler machine language display debuggers are described in [Jonsson] and [Zimmerman].

4.2 MEND

MEND [Berez] is a display debugger for MDL, a LISP-like language [Galley MDL]. It has typical debugging features such as breakpoints and stepping. Its display scheme is quite nice, showing substitution of values when the LISP-like function calls of MDL are evaluated. But MEND does not show the user code in context, as DIDL does. MEND runs in a separate process from the program being debugged, a feature provided by MDL. MEND exists as an extension to the MDL editor, and thus interfaces nicely with the rest of the MDL system.

4.3 Teitelman's Display-Oriented Programmer's Assistant

[Teitelman] describes a "Programmer's Assistant" for LISP which includes debugging capabilities. The Programmer's Assistant is a complete programming system based on the use of a bit-map display terminal. The display software implements display windows that are not only subsections of the display screen, but

which can also overlap and partially occlude one another in a manner akin to pieces of paper on top of one another. A window can be "brought to the top" to look at it if it is obscured. Since windows can be stacked in this way, there is much less of a problem of running out of display space or forcing a program to store text explicitly for possible redisplay.

Instead of using a keyboard, most of the commands in the Programmer's Assistant are invoked by pointing at them using a "mouse" [English] and its associated cursor. Only text need be entered through the keyboard.

Teitelman's system is a complete programming environment for LISP, containing an editor, debugging features, a documentation system, and a record of the typescript of a terminal session. The typescript itself can be read, edited, and operated upon to repeat, redo in a different way, or undo a previous action. The debugging features provided are not as complete as those provided by DDT or even DIDL, but are fairly extensive. However, Teitelman's main objective is not just to provide a debugger, but to present the user with a complete programming system, allowing him or her to switch easily between tasks, suspending one operation to do another. This flexibility enhances the user's debugging ability considerably.

The display operations the system provides are mainly concerned with selecting items from menus, editing typescripts and other output, and rereading portions of text. Though the window mechanisms allow random access to what is in a window, the method of input and output in a window is still fundamentally serial. The key omission is that the system does not point at information on the screen to indicate something to the user; only the user does the pointing. For instance, DIDL indicates the current LISP form being evaluated by pointing to it in the context of the code the form is found in. ESP does the same thing for machine language. Teitelman's system does not provide such a capability and does not seem oriented toward it. The system does not fully understand the text it has displayed, though it does know where it puts things, to allow menus and its editor to work.

4.4 COPILOT

A comprehensive system called COPILOT which comes closest to DIDL's philosophy is described in [Swinehart]. COPILOT is described as a complete "Interactive Programming System". The tasks of editing, compiling, debugging, and running programs are all integrated together in the system, as in Teitelman's system. The program, its data environment, and execution environment are all displayed. To accommodate all this information several display screens are used at once. COPILOT is designed around a small ALGOL-like language called MISLE. MISLE is a

compiled language; after an edit is made recompilation takes place automatically. COPILOT is largely written in MISLE itself, and most COPILOT commands are abbreviations for calls on MISLE procedures.

Like Teitelman's system, COPILOT consists of several processes for doing different tasks which the user can switch between at will, suspending one operation to do another. COPILOT also provides multiple process capability for user programs as well. But unlike Teitelman's system the concept of "non-preemption" is emphasized in COPILOT. "Non-preemption" means that the user does not have to stop a running task when he or she wishes to do something else. An example of the philosophy of non-preemption is that the user may edit while waiting for a breakpoint. When the breakpoint occurs, the user may choose to ignore it temporarily. Essentially, the user does not need to interrupt one system task in order to do another, and correspondingly, the system does not forcibly interrupt the user's work when something requiring the user's attention occurs.

4.5 Conclusions

Both Swinehart's and Teitelman's systems try to be all things to all people for a particular language. Their debugging features are just a part of the capabilities they must provide. As a result of their attempt at comprehensiveness they are perhaps not currently as fancy in their sub-parts as a system composed of more diverse elements. But it would be only a matter of time and effort before they could achieve the same level of sophistication as current less-integrated but fancier systems.

The advantages of working in such a unified system are clear: there is one clear design philosophy, and one method of doing things. So the user or programmer need only learn one (large) system, which is a considerably easier task than learning an editor, a debugger, an operating system command language, and so on. The danger in sticking to a unified system is in making the overall design too restrictive or limited, so that a user is unable to do what he or she wants to do. So such a system must always be open-ended in some way, to allow enhancements and additions to its components.

5. Basic Capabilities of DIDL

DIDL's features can be divided into two main groups: those that perform regular debugging functions, and those concerned with using DIDL's display capabilities. The display features are used when invoking the other commands, and so will be described first.

5.1 DIDL displays

The DIDL display screen is divided into three parts, from top to bottom: the Window, the Info-Line, and the Echo-Area. The Window is the largest screen area, and is used for displaying user code, stack frame traces and other possibly large things to print. The Info-Line is a single line just below the Window which gives some information about what is displayed in the Window. The Echo-Area is where user command type-in is echoed, where transient information is printed and where the running program and LISP are forced to direct their normal terminal output. (See Figs 1 and 2.) This method of screen organization is patterned after EMACS [Stallman], a display editor used extensively on MIT's ITS systems, where DIDL runs. (For an introduction to EMACS, see [Cicarelli].) EMACS was selected as a paradigm because most ITS users are very familiar with it or similar editors. There was no need to be different because the EMACS paradigm is generally excellent, and its use would contribute to the ease of using DIDL.

The two things most commonly displayed in the Window are user code (always LISP functions, "ground" or "prettyprinted" with indentation to show structure) and stack frame traces (similar to LISP "backtraces"). For user code (Fig. 1) the terminal's cursor is positioned at the left parenthesis of the current form. For stack frame traces (Fig. 2) each stack frame is displayed on exactly one line; the cursor is positioned at the left side of the currently selected frame.

If the user code or stack frame trace is too large to fit completely in the Window, a portion of it is displayed, and the Info-Line indicates this by giving the line number of the top line in the Window, and telling whether there is more undisplayed material following. Commands exist for moving to the next or previous Windowful of text, and for displaying the first and last Windowful. In this way the user can easily "thumb through" a display.

When user code is being displayed, the DIDL user may move around in the displayed code with commands similar to those in an editor, moving textually forward or backward to the next or previous left parenthesis, or moving structurally forward or backward to the next or previous form and its corresponding left parenthesis. The form moved to becomes the current form.

Fig. 1. A DIDL User Function Display

```

(LAMBDA
(SX SCREEN START-POS IN-FUNCTION INDENTATION)
(VARS
(LEFTPARENLOC RIGHTPARENLOC HT-ENTRY INDENT-COLUMN NEXT-POS)
(SETQ HT-ENTRY (HT-ENTER SX))
(SETQ LEFTPARENLOC (CREATE-LEFTPARENLOC))
(SETQ RIGHTPARENLOC (CREATE-RIGHTPARENLOC))
(RPLACD HT-ENTRY (LIST IN-FUNCTION LEFTPARENLOC))
(STORE-LEFTPARENLOC^BEGIN-POS LEFTPARENLOC START-POS)
(STORE-RIGHTPARENLOC^BEGIN-POS RIGHTPARENLOC START-POS)
(STORE-LEFTPARENLOC^SX LEFTPARENLOC SX)
(STORE-RIGHTPARENLOC^SX RIGHTPARENLOC SX)
(SETQ INDENT-COLUMN (+ INDENTATION (2ND (SETQ NEXT-POS (POS+1 START-POS))))))
(PUTSCREEN-CHAR SCREEN START-POS (CONS 50 LEFTPARENLOC))
(DO
((SX-LEFT SX (IF (ATOM SX-LEFT) NIL (CDR SX-LEFT)))
(ELEMENT)
(ELEMENT-LOC)
(ATOMIC-CDRP)
(LAST-ELEMENTP)
(ELEMENT-INDEX 0 (1+ ELEMENT-INDEX)))
((NULL SX-LEFT))
(SETQ ATOMIC-CDRP (ATOM SX-LEFT))
(SETQ ELEMENT (IF ATOMIC-CDRP SX-LEFT (CAR SX-LEFT)))
(SETQ LAST-ELEMENTP (OR ATOMIC-CDRP (NULL (CDR SX-LEFT))))))
DIDL Function: PUT-BLK-INDENT [Top line displayed: 1 ] --More--
x
Function: put-blk-indent Grinding PUT-BLK-INDENT.
ini15n5nbn
Breakpoint name: before-do

```

Fig. 2. A DIDL Stack Frame Trace Display

```

1: (DIDL-EVALHOOK-STEP ((/$ (SIND X) (COSD X))))
2:TRUEANOM: (TAND (/ $ THETA 2.0))
3:TRUEANOM: (*$ 1.01686 (TAND (/ $ THETA 2.0)))
4:TRUEANOM: (ATAND (*$ 1.01686 (TAND (/ $ THETA 2.0))) 1.0)
5:TRUEANOM: (*$ 2.0 (ATAND (*$ 1.01686 (TAND (/ $ THETA 2.0))) 1.0))
6:SUN-POSITION: (TRUEANOM MEANA)
7:SUN-POSITION: (SETQ MEANA (-$ (*$ TIME 0.9856) 2.4832) GUESA (TRUEANOM MEANA)
8:SUN-POSITION: (PROG (MEANA GUESA ECCEA TRUEA LAMBDA OMEGA OBLIQ PHI THETA GHAG
9:SUN: (SUN-POSITION (+$ (JULIAN DATE) (HHMMSS HOURS)))
10:SUN: (SETQ SUNPOS (SUN-POSITION (+$ (JULIAN DATE) (HHMMSS HOURS))) SKYANG (S
11:SUN: (PROG (SUNPOS SKYANG) (SETQ SUNPOS (SUN-POSITION (+$ (JULIAN DATE) (HHM
12: (SUN (QUOTE (45 45 0)) (QUOTE (45 45 0)) (QUOTE (1978 5 17)) (QUOTE (1 1 1)

```

```

DIDL Frame display [12 frames]
Stepping ;Stepping ;Stepping cStepping ;Grinding TRUEANOM.
Am inside the current form: MEANA => 1211.81776
Stepping ;Am inside the current form: 2.0 => 2.0
Stepping ;Grinding ATAND.
Stepping ;Am inside the current form: 1.01686 => 1.01686
Stepping ;Grinding TAND.
Stepping ,Returned: 605.90888
Type a char to continue:
Stepping f/fd

```

Similarly, there are commands for moving up and down stack frames in the stack frame trace. Each frame of the trace is numbered and shows the LISP form for which the frame was created, or as much as will fit on one line. In addition, if the form is in a user function, the name of the function is also displayed. The user can select such a stack frame, and have the function displayed, with the frame's form indicated by the cursor.

Stack frames and forms are shown in context, as if one were in a display editor. In general, DIDL attempts to display the context something is found in whenever possible. With a display editor, the context provided obviated the need for a listing, and DIDL attempts to do the same thing.

DIDL allows any user function to be displayed for perusal and possible action, such as setting breakpoints. And the user may return at any time to displaying the form that is being stepped or was broken on because of a breakpoint.

5.2 DIDL's debugging capabilities

DIDL's debugging features fall into three categories: frame environment examination and manipulation, breakpoints, and stepping. The general powers of the commands are given below; see appendix 1 for a complete list of commands. The features provided in DIDL include those found in other MACLISP debugging aids, such as TRACE [Moon], STEP [Rich STEP], DEBUG* [Waters], STEPR [Baron], and STEPPM [Morgenstern].

There are four ways DIDL is entered:

1. The user may invoke DIDL at LISP's top level. He or she can then examine functions and set, modify, and clear breakpoints. But since there is nothing pending on the stack, it is meaningless to look at stack frames or to single-step programs. After using DIDL, the user may quit out of it, and use LISP's top level just as he or she would normally. But the mechanism for detecting breakpoints will have been enabled.

2. If a user program is running and causes a LISP error, or if the user deliberately interrupts the running program, then the MACLISP system enters a "LISP breakpoint loop". The user is able to type at LISP as if he or she were at the LISP top-level, and so may start DIDL. Examining functions and changing or setting breakpoints is possible as in 1., and now in addition the user may examine the stack frames and environments of the interrupted program. However, it is not possible to continue the user program except by forcing a selected stack frame to

return, and stepping is also not possible. This is because the user program has been interrupted not by DIDL, but by a LISP breakpoint (not a DIDL breakpoint), and DIDL is unable to force continuation from such a breakpoint.

3. When a breakpoint is hit, DIDL is entered automatically. At this point the user may do everything mentioned in 2., and may also continue the program or single-step it.

4. When single-stepping a program, DIDL is invoked for every non-atomic form to be evaluated (EVAL'd). This case is similar to 3.

DIDL's stack frame commands allow one to select a particular frame using the display operations, and then perform LISP operations in the data environment available to that frame. The user does this by asking that a LISP form be evaluated in the environment of the current frame. This is useful for examining variable bindings, such as function arguments. Bindings can also be changed merely by evaluating (SETQ ...) in the environment. And any LISP function can be called in that environment, using the bindings from it. In addition, the user may force a particular frame to return a user-supplied value immediately.

The breakpoint setting and modifying commands require that a function definition be displayed. This happens automatically when stepping or when a previously set breakpoint is hit; the user may also explicitly ask for a function to be displayed. After the user selects a particular form in the function, using the moving-around commands mentioned earlier, he or she may set, clear, or rename a breakpoint at the form. In addition, the user may give a breakpoint a Condition, an Action, or a Patch.

A Condition is a LISP form to evaluate when the breakpoint is hit. If the Condition evaluates to NIL, no break occurs, but if the Condition is non-NIL, a break will occur, and DIDL will be entered. When a breakpoint is first set, its Condition is set to T, so that it will always break.

An Action is also a LISP form to evaluate whenever the breakpoint is hit. The Action is always evaluated, even if no break to DIDL occurs because of a NIL Condition. The Action always occurs before any break to DIDL takes place. Initially, a breakpoint's Action is set to NIL, which does nothing.

A Patch, if it exists, is yet another LISP form to evaluate. Its value is returned instead of the value generated by evaluating the form at the breakpoint. If the breakpoint breaks to DIDL, the Patch is not evaluated until after the user continues from the breakpoint, unless the user issues a stepping command after the break

occurs. In the latter case, the user steps the Patch and not the form at the breakpoint. The user is warned whenever a breakpoint with a Patch breaks to DIDL. Initially, there is no Patch on a breakpoint.

A "tell-about" command lists the Condition, Action, and Patch on the current breakpoint. All breakpoints have names, given automatically by DIDL (of the form BPTn, where n is a positive integer), or given explicitly by the user. There is a command to list the names of all breakpoints, and another command shows where a given breakpoint is set by displaying the form it is set on in the context of the containing function.

Breakpoints or stepping leave the program about to evaluate the form at the place at which the program was interrupted. This form is called the form that was broken on, regardless of whether it was gotten to by a breakpoint or by stepping. There are several commands to continue from stepping or a breakpoint. One command simply resumes execution where it left off. Another evaluates the form broken on, shows its value, and then continues. And yet another command "steps deeper" by evaluating the form broken on, and also breaking on any subforms in that form. DIDL does not bother to break when evaluating atomic forms, but does display their values.

There are several miscellaneous commands, including a help command and commands to clear the Window or the Echo-Area.

6. How DIDL works

6.1 Displaying user code

The most important part of DIDL's display capability is its ability to display LISP user code. There are two possible ways to do this: (1) display the source text of the original LISP code, or (2) display the internal form of the LISP code, printed out in human-readable form, as is done by PRINT, GRINDEF [Moon], etc. It seems at first that using the original source text is the obvious choice, because it is what is edited by the user, and is certainly the representation he or she is most familiar with. It may also contain comments (generally discarded by MACLISP when the program is read in) which would be helpful to see during debugging. However, in many cases the source text is not even close to the eventual internal LISP representation. Indeed, sometimes functions are defined by other functions and therefore do not exist at all in the source text. And the now common use of LISP reader macros may make the source text unreadable as regular LISP. For these reasons it was decided to display the prettyprinted (or "ground") internal representation instead of the source code.

When the source code is ground and displayed, DIDL needs to know the screen locations of all the forms displayed so that it can point to them using the cursor. Therefore the grinder must not only prettyprint the user code but must also store information about where it has decided to print things on the screen.

In order to save time and effort, the grinder grinds a particular user function only once, and instead of printing it, stores the text that would have been printed in a data structure called a Screen (something of a misnomer; [Swinehart] calls similar structures Scenes). A Screen is a virtual display screen. It is a vector of Lines, and contains enough Lines to store the entire ground function form. Each Line is a vector of characters, and contains exactly what would be displayed on a line of the display terminal. Usually only part of a Screen fits into the Window at a time; the various Window commands essentially scroll the current Screen back and forth under the Window.

Every user LISP form can be located by specifying the Screen it is in, and the Line and Column (position within a line) coordinates of its left and right parentheses. This collection of information is called a Loc (short for Screen Location). The Loc of a form needs to be looked up for two reasons:

1. To move around in a user function display, DIDL needs to know the Locs of the parent, the offspring, and the siblings of the current form. For instance, if DIDL knows the Loc of (+ (CAR A) B) in

(*(+(CAR A) B) (- C D))

then it also needs to know the Locs of (CAR A), (- C D), and (*(+(CAR A) B) (- C D)). (At present DIDL does not need to know the Locs of atoms in order to work. However, it would be necessary to know these Locs in order to implement some fancier features not currently present in DIDL.)

2. It is necessary to be able to look up the Loc of any arbitrary user form so it can be displayed. This is because the breakpoint and stepping mechanisms give DIDL only the form broken on, and no other information.

The second requirement was met by entering every form the grinder encounters into a hash table, called \$Hash-Table, along with its Loc. Thus when an arbitrary user form needs to be displayed in context, it is looked up in \$Hash-Table. Its corresponding Loc is then used to select the proper section of the proper Screen needed in order to display the form.

The first requirement was considerably harder to solve. The first scheme thought of was to generate a list structure, containing Locs, that mimicked the structure of the user function form it corresponded to. Such a structure is essentially a parse tree of the form. Moving around in the user form would be accomplished by traversing this tree structure with a recursive procedure, recursing to go deeper, and returning to go to shallower forms. This type of structure was found to be difficult to construct, and its access method was clumsy and inflexible. So a second scheme was devised, which was also a mimicking tree structure, but which contained back-links. This structure could be accessed iteratively because the back-links permitted going higher and lower in the tree without recursion.

Both these schemes were somewhat clumsy, and were expensive to generate in terms of time and storage, since they each generated something identical in structure to a user function, but several times larger. Fortunately, an orthogonal requirement made a new scheme come to light:

Besides moving in and out and forward and backward in user code, it seemed reasonable to be able to let the user jump directly to a place of interest. The most reasonable way to do this was to add a searching command, such as those found in text editors. Text searching seemed like a far better idea than searching the list structure of a user function directly, because to implement a flexible list searcher would have been a great deal of trouble. In addition, the text of a function already existed in its Screen. But in order to go from text in a Screen to a user form, some extra information had to be stored in each Screen. At each left and right parenthesis a special Marker was stored which contained not only the character to print but also

the form and Loc corresponding to the text form. A search command would, after finding the string being searched for, look for the nearest Marker to find out what form it was in.

It was quickly realized that the presence of Markers in a Screen made the mimicking data structures superfluous. The necessary linking of Locs to find parents, offspring, and siblings is implicit in the textual representation in the Screen! For instance, if the cursor is at a left-parenthesis Marker, and one wishes to go deeper into the list structure, it is only necessary to search forward for another left-parenthesis Marker. If one encounters a right-parenthesis Marker first, it is impossible to go deeper in the form. By using other equally simple searching algorithms, all the desired moving-around capabilities can be implemented. Indeed, the EMACS editor [Stallman] implements its list-oriented editing commands in this way.

So all necessary information about "what forms are displayed where" can be stored and accessed using only Screens with Markers and the \$Hash-Table. It is true that moving around in a Screen to determine list structure takes more time than using the mimicking structure schemes, but this is more than offset by the time and space saved by not having to generate the mimicking structures.

Essentially, it is easy to traverse the tree represented by a LISP list by locally parsing the printed representation of the list. This is what DIDL's moving-around algorithms implement. For a language with a more complicated syntax than LISP, this would be much harder to do, and so a parse tree would really have to be generated.

In summary: DIDL's grinder puts the textual result of grinding into a Screen for display later. Every form the grinder encounters is entered into \$Hash-Table, along with that form's Loc. A Loc gives the Screen the form is found in and the coordinates of the form on the Screen. At left and right parentheses on a Screen, special Markers are stored which give the Loc and form that the parentheses correspond to.

6.2 Stack frame examination and manipulation

DIDL's stack frame capabilities are made possible by a MACLISP function which gives information about frames on the run-time stack. This function, called EVALFRAME [Moon] [Steele], makes it unnecessary to look at the internal representation of the stack. EVALFRAME is a powerful debugging aid just by itself, and it makes writing debuggers much easier.

LISP creates a new stack frame each time it needs to evaluate an interpreted LISP form. For instance, in evaluating $(+ (* A B) C)$, a frame to evaluate $(+ (* A B) C)$ is created, then a frame for $(* A B)$, and then a frame for A . The frame for A returns with whatever A is bound to, and then a frame for B is created, and so on.

EVALFRAME allows looking at the stack, starting with the top frame and going downwards, one frame at a time. The code that contains the call to **EVALFRAME** will be the first frame returned, because it is the top thing on the stack. To get the top frame on the stack, one does an **(EVALFRAME NIL)**. This returns four items. The first indicates the type of frame (whether it was created by **EVAL**, **APPLY**, or by a LISP error handler) and is unimportant to **DIDL**. The second item returned is a Pdl-Pointer, and indicates the position of this frame in the stack. A subsequent call to **EVALFRAME** giving the Pdl-Pointer as an argument returns information about the frame below the frame referred to by the Pdl-Pointer. The third item is the form to be evaluated in this frame. The fourth item is called an A-List Pointer, and is essentially a handle on the data environment available to the stack frame.

When **DIDL** is entered, it immediately creates a vector containing the results of **EVALFRAME** calls all the way down the stack. This stack representation is edited while being built to remove references to **DIDL** and its associated functions, which are extraneous to the user's debugging needs. Then **DIDL** runs down this vector, looking for calls to user functions, and grinds any such user functions it finds, if they haven't already been ground. A call to a function always precedes evaluation of any forms in that function, and so a call to a function will always be found on the stack below any frames created to evaluate forms in the function. Since the **DIDL** grinder enters all forms it encounters into **\$Hash-Table**, it is guaranteed that any user forms on the stack will be in **\$Hash-Table** after the grinding is finished, and so **DIDL** will be able to display them in context, using the Screens they are found in.

To evaluate forms in the environment of a particular stack frame, **MACLISP**'s **EVAL** function optionally takes an a-list pointer, which specifies the environment in which to do the **EVAL**. And **MACLISP** provides a function called **FRETURN** which takes a pdl-pointer and a value to return, and forces the stack frame specified by the pdl-pointer to return with the given value. Both these mechanisms are used by various **DIDL** commands.

6.3 Breakpoints and stepping

Breakpoints and the stepping commands of **DIDL** both use the **EVALHOOK** mechanism of **MACLISP** [Steele]. Whenever an interpreted form is to be evaluated, LISP calls an internal **EVAL** function, which normally creates a stack frame and then

evaluates the form, calling itself recursively if necessary. The EVALHOOK mechanism allows this process to be controlled by the user.

The EVALHOOK mechanism is complicated, and will not be described in full detail here. Essentially, it allows a user-supplied function, which we will call the Hook-Function, to substitute for the internal EVAL function, and expects the Hook-Function to return the value of the form to be evaluated. Unless the Hook-Function desires to simulate the actions of LISP completely, it will eventually call EVAL to get the value of the form. But before and after it calls EVAL, it can do anything else it wishes to do, such as printing the form to be evaluated, or as in DIDL, invoking a debugger. And in addition, when the Hook-Function calls EVAL, it can control whether or not it wishes the EVALHOOK mechanism to be enabled when form's sub-forms are evaluated.

When a user sets a DIDL breakpoint, the form to be broken on is added to a list called the \$Breakpoint-List. When the user leaves DIDL, and starts or continues to run his or her program, the EVALHOOK mechanism is automatically enabled. Thereafter, whenever a user form is to be evaluated, the Hook-Function is invoked. The Hook-Function sees if the form it has been given is on the \$Breakpoint-List. If it is, then the Hook-Function executes the breakpoint's Action, and then invokes DIDL, if the breakpoint's Condition is non-NIL. When DIDL is entered, it announces to the user that a breakpoint has been hit. DIDL or the Hook-Function take care of returning the right value, and checking for a breakpoint Patch. If no breakpoint is found, the Hook-Function simply EVAL's the given form, enabling the EVALHOOK mechanism so that breakpoints in deeper forms can be found.

When the user is single-stepping, the EVALHOOK mechanism is also enabled. The Hook-Function checks for breakpoints in the same way, but if none are found, it breaks to DIDL anyway with the form to be evaluated, and DIDL tells the user that he or she is stepping.

This method of setting breakpoints does not modify the user's code in any way. Most other debuggers set breakpoints by changing the code at the place where the breakpoint is to be set, and must be careful to change the code back when the program is not running so that the user will not see the breakpoint code. If the debugger somehow loses the information about where it has set breakpoints, and leaves some in the user's code, then the code may be damaged in some sense, and may not continue to work, with or without the debugger. DIDL's "non-destructive" method of setting breakpoints does not suffer from this problem, though it does therefore have to supervise the running of the program continuously, which takes more time than letting the program run freely.

Dynamic watching for conditions, such as a variable's binding being changed, has not yet been implemented in DIDL. However, adding such features only requires adding more kinds of checks in the Hook-Function. Since the Hook-Function allows complete control over the running of a program, nearly any conceivable debugging feature can be added merely by having the Hook-Function do more work.

7. Implementation notes

7.1 The single process problem

One of the main problems DIDL encounters is that it must exist in the same environment as the program it is debugging. In particular, it must use the same stack as the user's program, and must share the same global naming environment. The latter problem is not much of a nuisance if unusual names are used by DIDL for its function and global variable names. However, the former problem is much more serious. DIDL cannot be a program on the outside, looking in at the user program while it is running. Instead, a fresh invocation of DIDL must be used each time it is entered. For instance, while single-stepping user code, each evaluation of a deeper form is done by a deeper invocation of DIDL. DIDL essentially calls itself recursively in an indirect way by use of the Hook-Function.

Multiple invocations of DIDL mean that static information, such as what is currently displayed on the screen, must be kept in global variables, and considerable care must be taken to make sure these variables are kept accurate. DIDL must remove the stack frames that are DIDL-related when creating the stack frame trace for the user. And if the user is not careful, or is using some MACLISP features that DIDL is also using, then the user can accidentally destroy some of DIDL's environment, or make it impossible to use DIDL.

These problems would all be solved if DIDL were able to run in a separate process and environment from the program it was debugging. Probe [Honeywell] runs on the Multics user's stack, and suffers from similar problems. TOPS-10 DDT [DEC] runs in user's memory space, and can be easily destroyed by a user program gone wild. But many other debuggers, such as ITS DDT [MIT], and BDDT [BBN BCPL] use the multiple process capability of the operating systems they run under to prevent the program being debugged from damaging them in any way. COPILOT [Swinehart] makes extensive use of multiple processes for all its operations, and has the capability of debugging multi-process user programs.

The MIT LISP Machine [Bawden], which is a special processor specifically designed for running LISP, implements multiple processes with a concept called a "stack group". The LISP Machine debugger runs in one stack group, and can debug a program running in another stack group by examining its stack and controlling its execution.

7.2 Coding

DIDL was coded in a mostly bottom-up fashion. The grinder was written first, and was by far the most difficult part of the program to write. Several different grinding schemes were partially coded and then discarded because of their complexity. Once the grinder was finished, the low level display operations were coded. Then the top-level and the command reader were written, and the stack frame trace display was coded. At this point, the simpler method of storing Screen position information using Markers, as described in section 5.1, was thought of, and so the grinding routines were redone and somewhat simplified. Finally the rest of the commands were added. Surprisingly few recalcitrant bugs appeared; the most annoying ones had to do with enabling and disabling the EVALHOOK mechanism, but these were fixed fairly quickly. Most of the low-level functions worked perfectly the first time because of their simplicity.

The author has had experience maintaining and expanding several very large programs, but has not written many medium-sized systems from scratch. Once the initial hurdle of the grinder was cleared, the coding went very quickly. A fair amount of time was spent designing and redesigning the data structures that DIDL uses; they are carefully documented, and DIDL tries to be consistent when using them. This care paid off, and made DIDL relatively easy to write and debug.

8. Future work

8.1 Dynamic watching

The major feature specified for DIDL which has not yet been implemented is the ability to watch for some arbitrary condition occurring while the user program is running. The programmer should be able to break to DIDL when:

1. An arbitrary condition specified by the user becomes true.
2. A particular atom is evaluated.
3. The binding of a symbol is changed.
4. A form evaluates to a particular value.
5. A form matching a particular pattern is encountered.

Adding these dynamic capabilities is not difficult, and involves only adding things for the Hook-Function to look for and adding the commands necessary for the user to control these features.

In addition, DIDL should be able to display the values of selected forms continuously while the program runs, updating the display whenever the values change. Such a feature is present in RAID [Stanford], and is quite useful.

8.2 Debugging data as well as code

LISP programs frequently create large, complicated data structures which may take a considerable amount of run-time to generate. To be able to display data, move around in it, and have the ability to patch or modify it, in a manner similar to DIDL's method of moving around in code, would be a great boon to many programmers. Currently, a program bug that mangles data may be easy to fix, but to fix the data without recreating it from scratch is frequently much harder.

8.3 Complete interactive programming systems

There are two combined MACLISP-text-editor systems which provide for simple and transparent transfer between LISP and an editor: LISPT [Kulp] and LEDIT* [Rich LEDIT*]. The packages provide features such as automatically reloading an edited function and allowing the user to specify an individual function instead of just a text file to edit. A debugger could be incorporated into such systems allowing the user to edit, insert patches, set breakpoints, debug, and run programs, all in the same text display environment. Such a system would be leading up to kinds of capabilities

provided by the Programmer's Assistant [Teitelman] and COPILOT [Swinehart].

A combination of DIDL and LEDIT* or LISPT is still far from these comprehensive, integrated systems. However, it would be not be extremely difficult for a LISP-editor package and DIDL to know about each other, switch between themselves, and exchange information, making life for the programmer that much easier.

8.4 Human engineering

DIDL usability could be enhanced in several ways. The display management in the Echo-Area could stand some improvement, as the typescript in the Echo-Area frequently becomes messy and garbled. Some commands could be expanded to do more than one thing, so that DIDL could second-guess the user in some cases. For instance, there is presently one pair of commands for moving up and down frames in a stack frame display, and another pair for moving to next and previous forms in a user code display. Each pair of commands should duplicate the functions of the other pair, or the two pairs should be merged into one, so that the user need not remember different commands for similar actions.

DIDL currently will run on any display terminal that supports random cursor positioning and selective erase capabilities. However, it does not use features which fancier display terminals can support, such as drawing lines or highlighting text using reverse video; use of such features would make DIDL more pleasant to use.

9. Conclusions

9.1 Context displays

The most significant advantage that DIDL has over conventional debuggers is that it displays the code or information in which the user is interested in context. The context obviates the need for a separate reference copy of the program and allows the user to work using only the display terminal. More importantly, the context provided encourages the programmer to think about a suspected bug not in terms of a single program statement, but instead while considering the actions of a whole section of a program.

9.2 Understanding displayed text

DIDL is a demonstration of the fact that it is possible to write programs that use display terminals to full advantage. By displaying textual objects and understanding what is displayed where, DIDL can point to and manipulate whole objects, and not just text. Almost any text that is displayed has some kind of structure; it is not just a collection of characters. Understanding and using this structure is a key part of using display terminals fully.

There are many graphics software packages that allow manipulation of complex objects easily. The techniques for doing this manipulation are fairly well understood. But for complex textual displays, there are no standard ways of doing comparable kinds of manipulation yet, and many good ideas probably remain waiting to be discovered. Hopefully, DIDL and other programs will contribute something toward this goal.

Appendix I - DIDL Commands

(didl) enters the debugger. It is entered automatically when stepping or when a breakpoint is hit.

n indicates an optional positive or negative numeric argument, typed before the command. 1 is assumed if no argument is given.
- means -1.

form indicates the command prompts for a form to be typed in.

func indicates the command prompts for a function name.

bpt indicates the command prompts for a breakpoint name.

Window Manipulation

< First window.

> Last window.

n V Next or previous window.

L Redisplay.

(Show left parenthesis of current s-expression.

) Show right parenthesis.

Frame Display

F Show the stack frames.

n U Go up a stack frame.

n D Go down a stack frame.

n J Jump to stack frame n.

. Select the current stack frame and display the user code it corresponds to, if possible.

Looking at function definitions

X func Examine a user function.

n I Move forward to the left parenthesis of the next form, no matter what depth it is at, making that form the current form. (In)

n O Move backward to the previous form. (Out)

n N Move forward to the next form at this depth. (Next)

n P Move backward to the previous form at this depth. (Previous)

G func Regrind a user function, even if it has been ground before.

Doing things in the current frame

E form Evaluate a form using the environment of the current frame.

R form Force the return of a given value from the current frame.

Stepping and breakpoints

/ Show where stepping or a breakpoint has interrupted the program.
 C Continue from stepping or a breakpoint.
 , Continue from stepping or a breakpoint, but show the value
 that will be returned before continuing.
 ; Step deeper, and show the value that will be returned.
 BS Set a breakpoint at the current s-expression.
 BC Clear the breakpoint at the current s-expression.
 BN bpt Set a breakpoint at the current s-expression, and name it the given
 name, or rename the current breakpoint.
 BI form Set a condition on the current breakpoint, which will break only
 if the condition is non-nil. (If).
 BA form Set an action on the current breakpoint. The action will always
 be done, even if the breakpoint does not break because of a
 condition.
 BP form Set a patch on the current breakpoint. The value of the patch
 will be returned instead of the value given by the form at the
 breakpoint. The form at the breakpoint will not be eval'd.
 BT Tell about the current breakpoint, listing its name, condition,
 action, and patch.
 BL List the names of all breakpoints.
 BG bpt Go to the given breakpoint, displaying where it is.

Miscellaneous

Q Quit from DIDL.
 ? Print a list of all commands, with short descriptions.
 ^K Clear the Window.
 ^L Clear the Echo-Area.
 <space>
 <cr>
 <lf> are ignored.

Appendix II - References

- [Baron] Robert V. Baron. Massachusetts Institute of Technology, Laboratory for Computer Science, internal on-line ([MIT-AI] LIBDOC;STEPR >) for the STEPR MACLISP debugger.
- [Bawden] Alan Bawden, Richard Greenblatt, Jack Holloway, Thomas Knight, David Moon, Daniel Weinreb. LISP Machine Progress Report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, AI Memo No. 444, August, 1977.
- [BBN BCPL] Bolt, Beranek, and Newman, Inc. BCPL Reference Manual, 1974.
- [BBN WE] Bolt, Beranek, and Newman, Inc. Internal documentation for the WE editor.
- [Berez] Joel M. Berez. A Dynamic Debugging System for MDL, Massachusetts Institute of Technology, Laboratory for Computer Science, TM-94, January, 1978.
- [Cicarelli] Eugene Ciccarelli. An Introduction to the EMACS Editor, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, AI Memo No. 447, January, 1978.
- [DEC] Digital Equipment Corporation. DECsystem-10 Assembly Language Handbook, 1972.
- [Deutsch] L.P. Deutsch and B.W. Lampson, DDT Time Sharing Debugging System Reference Manual, University of California, Berkeley, Document #30.40.10 (rev.), May 1965.
- [English] W.K. English, D.C. Engelbart, and M.L. Berman. "Display Selection Techniques for Text Manipulation", *IEEE Transactions on Human Factors in Electronics*, Vol. HFE-8, No. 1, March 1967.
- [Evans] Thomas G. Evans and D. Lucille Darley. "On-Line Debugging Techniques: A Survey", *Proceedings of the Fall Joint Computer Conference*, 1966, pp. 37-50.
- [Galley ESP] S.W. Galley, Debugging with ESP -- Execution Simulator and Presenter, Massachusetts Institute of Technology, Laboratory for Computer Science, Programming Technology Division Document SYS.06.01, November,

1971.

[Galley MDL] S.W. Galley and Greg Pfister. MDL Primer and Manual (for versions 54 and 104), Massachusetts Institute of Technology, Laboratory for Computer Science, 1977.

[Honeywell] Honeywell Corporation. Multics Programmer's Reference Manual, AG92A, Rev. 2, 1978, pp. 3-288 - 3-316.

[IBM] International Business Machines Corporation. IBM System/360 and System/370 Fortran IV Language, GC-6515-10, pp. 131-137.

[Jonsson] Sven Ingvar Jonsson. "On-Line Program Debugging", *BIT* 8 (1968), pp. 122-127.

[Kotok] A. Kotok, DEC Debugging Tape, Memo MIT-1 (rev.), Massachusetts Institute of Technology, December, 1961.

[Kulp] John L. Kulp. Massachusetts Institute of Technology, Laboratory for Computer Science, internal on-line documentation ([MIT-AI] INFO; LISPT >) for the LISPT LISP-and-editor system.

[MIT] Massachusetts Institute of Technology, Artificial Intelligence Laboratory. Internal on-line ([MIT-AI] INFO; DDT >) documentation for the DDT debugger.

[Moon] David A. Moon. MACLISP Reference Manual, Revision 0, Massachusetts Institute of Technology, Project MAC, April, 1974.

[Morgenstern] Matthew Morgenstern. Massachusetts Institute of Technology, Laboratory for Computer Science, internal on-line ([MIT-AI] LIBDOC; STEPMM >) documentation for the STEPMM debugger.

[Reiser] J.F. Reiser. BAIL: a Debugger for SAIL, Stanford University, Department of Computer Science, CS-523, October, 1975.

[Rich LEDIT*] Charles Rich. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, internal on-line ([MIT-AI] INFO; LEDIT* >) documentation for the LEDIT* LISP-and-editor system.

[Rich STEP] Charles Rich. Massachusetts Institute of Technology, Artificial

Intelligence Laboratory, internal on-line ([MIT-AI] LIBDOC;STEP >) documentation for the STEP debugging aid for MACLISP.

[Stallman] Richard M. Stallman. Internal on-line ([MIT-AI] INFO;EMACS >) documentation for the EMACS editor.

[Stanford] Stanford University, Artificial Intelligence Laboratory. Internal on-line ([SU-AI] RAID.PMP[S,DOC]) documentation for the RAID debugger.

[Steele] Guy L. Steele Jr., Jon L. White, and Howard I. Cannon. Massachusetts Institute of Technology, Laboratory For Computer Science, internal on-line ([MIT-AI] .INFO;LISP NEWS) documentation for MACLISP.

[Swinehart] Daniel C. Swinehart, Daniel C. COPILOT: A Multiple Process Approach to Interactive Programming Systems, Stanford University, Stanford Artificial Intelligence Laboratory Memo AIM-230, July, 1974.

[Teitelman] Warren Teitelman. "A Display Oriented Programmer's Assistant", *Fifth International Joint Conference on Artificial Intelligence*, August, 1977, Vol. 2, pp. 905-915.

[Waters] Richard C. Waters. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, internal on-line ([MIT-AI] LIBDOC;DEBUG* >) documentation for the DEBUG* MACLISP debugger.

[Zimmerman] Luther L. Zimmerman. "On-Line Program Debugging - A Graphic Approach", *Computers and Automation*, November, 1967, pp. 30-34.