LABORATORY FOR
COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-94

A DYNAMIC DEBUGGING SYSTEM FOR MDL

JOEL M. BEREZ

JANUARY 1978

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-94

A DYNAMIC DEBUGGING SYSTEM FOR MDL

Joel M. Berez

January 1978

MIT/LCS/TM-94

A DYNAMIC DEBUGGING SYSTEM FOR MDL

Joel Mayer Berez

January 1978

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

A DYNAMIC DEBUGGING SYSTEM FOR MDL

Joel Mayer Berez

Abstract

Program debugging is a time consuming process. Conventional debugging techniques and aids typically give the user a narrow view of the program's operation, making debugging difficult. A debugging system that would present a clear overall picture of a program's behavior and would be both flexible and simple to operate would be a valuable tool. Such a system was designed and implemented in and for MDL, a high-level applicative programming language. This report discusses: the design alternatives considered during the debugging system's design and implementation phases, the reasons for the resulting design choices, and the system attributes. A major attribute of the system (MEND) is that it does not simulate the program being debugged but instead monitors it from another process. This attribute results in a robust and viable debugging system, because MEND need not be modified in order to handle each new extension to MDL and/or each new user-defined primitive.

This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the Degree of Bachelor of Science.

Acknowledgements

I wish to thank Al Vezza, for supervising this work and guiding me along the road to winnage; Stu Galley, for the original idea; Bruce Daniels and Gerald Farrell, for laying some of the groundwork I have built upon; Brian Berkowitz and Chris Reeve, for patiently repairing my ailing MDLs; Marc Blank and Tak To, for support work and for providing me with company during all-night console sessions; and all of the other ITS and DynaMod hackers who have built a system well worth using.

This research was supported 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.

Table of Contents

I Introduction

I.1 Background

A time-consuming and frustrating aspect of computer programming is the debugging of faulty programs. Current debugging techniques involve tracing through the present operation of the program and mentally comparing its action with one's concept of what should be happening. With few exceptions, an understanding of where the program fails to conform to "correct" operation must be made before the cause of the failure can be determined and corrective action taken. This is where much of the difficulty occurs.

In conventional debugging, it is rare for the programmer to have available any more than the most basic aids. One usually has to extrapolate from a bare minimum of information (such as machine generated error messages) or one may be buried under a large excess of information, mostly irrelevant (such as a core dump). Even with the more advanced aids, the programmer typically gets but a small window into the operation of the program through which, sooner or later, she or he will locate the problem. A well-localized fault will be relatively easy to spot compared to a global problem that the programmer may only catch glimpses of through the debugging window.

In a compiler language, the programmer's best hope is to insert statements to print intermediate results or to try to separate the program into easier-to-handle modules. There are a few more advanced aids available[1], but their use is limited. One problem is that the program is, in effect, first translated into a lower-level language (generally "machine" language) and then executed interpretively in that language. The original symbols and syntax of the source program are lost, or saved only with great difficulty, making analysis and manipulation of the executing program a very painful process.

If the programmer is using an interpretive language with facilities for interaction, things are considerably easier. The common technique is to stop execution at strategic points and examine the state of the environment. Since this is done interactively, with the source program still available in more or less its original form, the cause of the problem can often be found in less time than it could otherwise. One of the best examples of this approach is the use of DDT (Dynamic Debugging Tool/Technique)[2,3].

DDT basically works with machine-language programs. However, by freely translating between numbers and symbols through the use of a table generated by the assembler, DDT makes programs look to the programmer like symbolic assembly-language programs. The user of DDT can operate in free-run mode or in n-step (statement) mode, switching between them at will. In either case one can set a breakpoint at any statement, which will cause execution to stop just before

the statement is executed. Whenever stopped in DDT, the user can examine or change the contents of any location. This can be done in several data modes (e.g., unsigned octal, full ascii, sixbit, etc.) including the use of symbols to represent addresses. Arbitrary numeric expressions can also be evaluated without affecting the program.

One main advantage of DDT is that the debugging environment is very similar to the language environment the programmer used to write the program. One has to learn only the DDT commands rather than an entirely new language. Another advantage is the user's ability to make changes in the program and data interactively at execution time, with ready ability for viewing the results of the changes. In addition, one can quickly see the results of the changes and act accordingly.

The main deficiency of DDT is that, although its name includes the word "dynamic," its operation is really static. The application program can run freely, but when the programmer wants to see what is taking place, the program must be stopped. Although the real interest may be about changes on a gross scale, perhaps thousands of program statements, if one does not know exactly where the program is misbehaving, one may be required to suspend execution of it every few instructions to examine variables in order to obtain a true picture of the program's behavior. Thus the programmer sees *not what is taking place*, but *what has taken place*, and through small windows at that. This is inefficient, and the programmer can become bogged down in detail that hinders the discovery of the true problem. The situation can be improved with the use of breakpoints that allow the program to execute freely until a breakpoint is reached, at which point the program halts. DDT is a powerful tool but still leaves much to be desired in a debugging tool.

ESP[4,5] (Execution Simulator and Presenter) is one solution to the static problem. It really is dynamic in that large amounts of data are constantly being displayed for the programmer while the program is being executed (actually, simulated). The information is presented in graphic form to improve readability and reduce confusion. A user of ESP may watch areas of the display where data of particular interest are being presented. One also has many options including control over the speed of execution, the type and quantity of data displayed, and special (more flexible than just a breakpoint) conditions for halting execution. In this way one can structure and control the picture presented to more easily understand what the simulated program is doing. And that is one key step in the process of debugging.

Like DDT, ESP has deficiencies also. These are mainly in the area of editing and DDT-like examination of a faulty program. DDT is a sophisticated language that ESP does not attempt to entirely replace. The flaw in ESP is that it is not compatible with DDT. Ideally both should be simultaneously available to the programmer, who can use features of each as the need dictates.

DDT and ESP work with a low-level language whose operation can be shown fairly simply. For example, ESP often shows flow of control by just displaying the actual section of program being executed and pointing to the current statement[6]. It also draws lines to show where branching has occurred and in some cases even indicates looping. This display philosophy can be readily extended to higher level languages that are line oriented, like BASIC, but it fails with applicative ones, like LISP or MDL. The latter type does not use a linear control flow, but uses a complex depth-first tree structure. Furthermore, quite complicated data structures can be built (or themselves executed) that bear very little relation to the appearance of the program.

A good basic display for MDL was used in MUMBLE[7] (Gerald Farrell's monitor and debugging aid). The code being executed is shown in stack form. Each line shows a piece of code being evaluated. As each object in the bottom line is evaluated, it is replaced by a single downward-arrow symbol in this line and then printed on a new line. In this way the evaluation can be followed from the top level down to the current object being evaluated. Furthermore, after the bottom is reached, the value returned by each line replaces its symbol in the previous line. With this mechanism, the programmer can follow execution in a natural and reasonably clear representation.

MUMBLE had some difficulties arising from the fact that it simulated execution rather than just watching it and letting it proceed naturally. This caused it to run slowly and to be complex and fragile. At the time MUMBLE was written, the MDL compiler was not yet perfected and the language itself lacked some of the multiprocessing features that would have made simulation unnecessary. Later Farrell replaced MUMBLE with a debugger utilizing new software related to single-stepping a process, which eliminated the simulation but also eliminated the feature reflecting results of an evaluation back into the original code. Also, a mode was added that allowed the programmer to attach conditions to parts of the program which would stop execution when and if a condition was false. This is as far as MUMBLE ever progressed, and it was not in use as of the time of the proposal for the current work.

I.2 MEND

After the MDL compiler became operational and many additional new software features became available, it appeared that it would be possible to design and implement a debugging system that would be comprehensive, easy to use, and reasonably fast. It was therefore proposed that a debugging system for MDL called MEND (Mdl Executor, aNalyzer, and Debugger) be designed and implemented. It has the following characteristics. (A glossary of special terms,

those in all capital letters, used here and throughout the rest of this memo may be found in Appendix B.)

1. MEND possesses a display similar to that of MUMBLE, including the replacement of arguments in a FORM by their values as they are evaluated.

2. Execution is monitored from another process (as opposed to being simulated as in ESP) using 1STEP and related features[8].

3. Execution speed is variable by the user including a static single or multi-step mode where desired.

4. MEND allows execution to run freely below a certain depth of evaluation or between certain points in the program and to run controlled elsewhere.

5. Unconditional and conditional breakpoints are available that can be attached to any object to halt execution before evaluation of that object.

6. The system is capable of keeping track of programmer specified conditions and of changing modes or giving some visible indication when the conditions are met (or not met).

7. Information, such as the local values of programmer specified ATOMs and the values of programer specified FORMs, is constantly displayed beneath the main display.

8. Each line in the main display area is &-printed (abbreviated printing, see glossary) and can be viewed in full at any time.

9. At any time, with execution stopped, the user can EVAL objects in the ENVIRONMENT of the program. This means the user can examine the state of the program or change it.

I.3 Possible Additional Characteristics

Certain other characteristics were seen as desirable for MEND but possibly beyond the scope of this project. If time permitted these features were to be included in the system:

1. The IMLAC (see section II.1) multi-screen capability would be used to allow the user to rapidly switch between the debugger display and the program's own output. Other system output could also be put on additional pages.

2. The editor IMED (an editor for MDL objects analogous to IMEDIT[9]) would be tied into the system to allow easy editing. PRINTTYPE and READ-TABLEs would be used to allow breakpoints to be easily set and removed in IMED as single symbols. Other control codes and statements could also be inserted using this editor.

3. At the applications programmer's option and within certain limits, execution could be reversed either so that something different could be tried or

for purposes of reexamining the process for something that may have been missed the first time. This feature could come in two possible forms, the UNDO package[10] to actually reverse execution or a simulation displaying information previously stored by the system.

I.4 Design and Implementation

MEND was designed with the intent of providing the application programmer with many options so that debugging could proceed in the most suitable manner for each situation. In the normal running state, MEND displays several kinds of information on the screen. Most important is an area showing the execution of the application program being debugged in stack form. The only other area that is always present is a line or two of status information about the current operation of MEND showing its current speed of execution (user adjustable) and the state of each modifiable mode.

It was intended that the output of the application program be saved by MEND for later reference. The user of MEND could then elect to have the most recent output constantly displayed in a window on the main screen (see section on future work). If multi-screening were available, the output could be kept on another virtual screen. That screen could be displayed or made invisible at the user's option without stopping MEND.

Information such as programmer specified values of ATOMs, structured objects, and, in general, the value of any MDL expression may be constantly displayed. MEND is also capable of displaying such information on an exception basis according to some predescribed condition. Such information is &-printed but is viewable in full when desired.

It is important for a debugging system like MEND to be compatible with and to take advantage of available software in related areas. One such area is editing. There were two MDL editors in use when the proposal for this work was made, EDIT[11] and IMED. The main difference between them is that IMED uses the local editing features of the IMLAC while EDIT does not. EDIT, however, has the advantage of being the only one that possesses breakpoint capabilities. Whichever proved to be most compatible with MEND (possibly both) would be slightly modified to allow the setting and removing of certain MEND codes including, in the case of IMED, breakpoints.

MDL itself has many features that greatly aid the debugging process. One of these is FRAMES. This function can be used to print the stack of functional evaluations and applications when execution is halted at any depth below the top level. At this point it is also possible to get the values of objects in the current ENVIRONMENT and to change them. One can even restart execution at a higher

level after making such changes. Because the MDL debugging features are quite powerful, MEND was designed to allow the user to stop execution (of the application program) and to use these aids or any others built in to MDL with the MEND system itself transparent. Evaluation would take place in the ENVIRONMENT of the application program.

MEND now includes the main features of all the debuggers that have been mentioned and enough other features that it should prove to be quite useful for the analysis and debugging of MDL programs. It should also serve as a good example of the type of debugging system that can be built around an applicative type language.

II Terminal Display

II.1 IMLAC Console Program

One basic concern throughout the project was the display: how the information made available by MEND would be presented to the user. To a large extent, the physical characteristics of MDL, ITS[12] (Incompatible Timesharing System, the operating system used on the Dynamic Modeling System computer), and the available terminals dictated what was reasonably possible.

The terminal most commonly used by users of the Dynamic Modeling System is the IMLAC PDS-1. This is a minicomputer capable of having programs loaded into it from the PDP-10 host computer. One program written for it by Dave Lebling is MSC, a multiple-screen terminal program. Up to four virtual screens (or pages) can be created that will individually operate like the actual screen area of the standard terminal program (SSV[13]). Output may be directed to any one of the screens and any screen may be visible or invisible at any instant of time, at the programmer's option. Selection of screen is controllable either by program from the host or locally by the user.

It was originally intended that MEND would use MSC for its normal display. One page would constantly show MEND's representation of the control stack of the application program. Output initiated by the program being debugged would go to a second page. A third page would be used for interaction with MEND and would show user typein along with any output from MEND that one was interested in seeing. The latter would include, by user request, full displays of both objects printed in an abbreviated form on the page containing the control stack of the application program and values being monitored or traced. The user could switch back and forth among the pages at will during the execution of the program. The application program would have a full standard screen to write onto, and ample room would be available for the information to be displayed by MEND.

After a fair amount of testing, this proposal was discarded for the following reasons. First, MSC was supposed to look just like SSV for individual virtual screens. Unfortunately new features added to SSV had not also been added to MSC. A primary reason for this disparity was that the new features encroached upon the IMLAC's character space. An SSV with all current features can only hold about one and a half full pages of text, where a page is the amount that can be visible at one time. The overhead required for additional screens reduces it still further. Therefore, with all features included, four virtual screens could each only average about one-third full. Without many of the current features, people were reluctant to use MSC.

The second and perhaps most devastating problem with MSC is that it is not properly supported by ITS as is SSV. Ordinarily the operating system will

keep track of where on the page the terminal's cursor is and will properly handle such updating as deletions even in the face of random access performed (if done by request to the system). When using MSC, ITS does not realize that information is being written onto more than one screen and will therefore often move the cursor to the wrong position.

MEND could completely control positioning for typeout and echoing on typein but that would add the large overhead of having to run a non-trivial routine for each character typed out on the display. Also, because MSC is not the standard console program, requiring the user to load it before using MEND might discourage the use of MEND. (It takes between about 30 seconds and a couple of minutes, depending upon system load, to load a new console program into the IMLAC.)

From the outset it was intended that MEND be used routinely by programmers as debugging problems arose. Therefore it was decided that the proper way for such a system to operate was to use, as far as possible, the common environment so as to keep the overhead for invoking MEND small. This philosophy, which had been seen to affect the success of many earlier projects, decided between the alternatives and in this case led to the final decision to use SSV instead of MSC.

II.2 Terminal Independence

The MEND terminal handling capabilities are actually quite general and MEND does not depend exclusively on SSV. For the purpose of dividing the screen area into several sections, horizontal lines are sometimes drawn (see Appendix A showing sample display). With an IMLAC and SSV these lines could be drawn quite simply using graphics mode. However, for purposes of generality with regard to terminals, these lines are instead formed by using underbar characters (on a line of their own). By using no actual graphics MEND can be used with almost any display terminal having random access. MEND outputs display commands, such as clearing a line, as escape codes to ITS which then translates these into the appropriate commands for the terminal in use. ITS currently knows about several types of display terminals in use at the Laboratory for Computer Science, and other types of terminals located at foreign sites on the ARPA network may be handled by interface software that simulates a known type. Naturally MEND can handle a large range of possible line and screen lengths. (The current version of SSV provides four possible character sizes.)

II.3 Control of Screen Sectioning

There still remained the question of how to handle the multi-sectioning of the displayed information. Originally three sections corresponding to the three virtual screens in the aborted MSC implementation were planned. The bottom section would hold user typein and application program output. Three possible methods of achieving this involved having ITS, MDL, or MEND do various amounts of the work with increasing overhead and decreasing speed for those three, respectively.

The most attractive solution utilized an ITS feature allowing the specification of an echo area at the bottom of the screen where echoed input would always be printed (with the echoing handled by the system, which is the normal case). After some experimentation this method of handling the typein and application program output was rejected because typeout and deletion are handled by MDL which ignores the echo area. MEND would effectively have had to control all typeout and monitor all typein, which would have made the echo area useless.

Experimentation with the second solution, an indirect method, involved monitoring of special MDL memory locations where information concerning horizontal and vertical page positions is stored. It was soon discovered that MDL becomes confused quickly, contains several bugs with respect to this position information, and generally has a poor idea of where it is actually printing.

The third solution appeared to be the most painful from an implementation and efficiency point of view. MEND would need to control the printing of every character on output and certain characters on input, constantly checking page positions by querying the system. Not only did this slow output, but also MEND was forced to constantly move the cursor to a safe position in case MDL managed to sneak some output past it, which it occasionally does.

Fortuitously the problem was neatly solved when a little known feature of ITS was discovered. It is possible to open a channel to the terminal in a mode that would cause all output to appear in the echo area. By creating this echo area and reopening MDL's normal terminal output channel in this mode, it is possible to cause MDL and the application program to think that the entire screen consists of only the echo area. All output including application program display escape commands is automatically routed by ITS to this echo area. MEND sends its output to a second channel opened in the normal mode, thereby allowing it to use an area of the screen unknown to and left untouched by the application program. The physical cursor stays where the last used logical cursor left it, thereby eliminating most of the unnecessary cursor movement, resulting in a more pleasing visual effect.

As a result of this solution, the screen is divided into only two main sections. All typein appears in the lower section, whether it is to the application

program or to MEND. Application program output also goes to the lower section, as does unstructured output produced by interaction with MEND. The upper section, in general, contains only those items that occupy one line of the display each. As will be explained later, these items include all output automatically displayed by MEND during execution of the application program.

III MSTACK: MEND's Representation of the Control Stack

III.1 Program Execution in MDL

MEND's primary role is to allow the applications programmer to visually monitor the execution of a program. In a language like MDL, this is most easily accomplished by showing the programmer a "picture" of the control stack.

A MDL program consists of the evaluation of a single object. The object is usually structured in some manner and itself contains other objects. The most common object is the FORM. This is a list of objects in which the first is (or evaluates to) some function and the rest are arguments to that function. A FORM is evaluated by applying the function to its arguments, usually after the arguments are themselves evaluated. This evaluation actually is initiated by the MDL interpreter by applying the function EVAL to the original object. EVAL takes an object as its argument and returns the value to which it evaluates.

Figure 1 shows a (simplified) static representation of the evaluation of a MDL object. Starting with the FORM (list of objects in angle-brackets) to be evaluated, the flow of control/evaluation may be described as a depth-first search through the tree pictured. The arrows represent values being returned to previous levels. At the end of this "search" the FORM returns the value shown at the top.

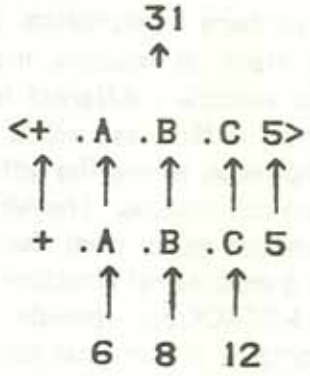

Figure 1

Typically the control stack will start with one object on it, the FORM to be evaluated. This evaluation will first require that the objects in the FORM (possibly FORMs themselves) be added to the stack and evaluated. EVAL recursively calls itself for this purpose. The stack builds (downward, by convention) until some object is placed on it which is known by EVAL to need no further evaluation. This object is returned as its own value to the previous level and values continue to

be returned upward until a level is reached where another object must be evaluated. In this manner the stack grows and shrinks until the topmost (initial) object returns a value.

III.2 Monitoring Program Execution

The manner in which the stack builds, the objects are evaluated, and the values are returned illustrate most of what the program is doing. Other factors, including side effects and compiled code, will be discussed at the end of this chapter. MEND's main display therefore shows a representation of the stack being continuously updated as execution/evaluation proceeds.

There are essentially two ways that MEND could follow the flow of control of the application program. The most direct way, as attempted by MUMBLE[7] and discussed in the last chapter, would be for MEND to execute the application program by simulating the operation of MDL. This type of simulation has been shown to require a complex and all too often fragile structure. The debugging program would need to be constantly updated to match changes and additions to the MDL language, but more importantly it would fail to properly handle programmer-defined primitives, several varieties of which are provided for by MDL.

A far more satisfactory method is to allow MDL to execute the application program in more or less its normal fashion but to stand back somewhere and watch. Fortunately MDL contains a mechanism ideal for this, called multiprocessing. Basically another control stack, or process, may be created (independent of the first) that may be used to execute a different function with its own set of variable bindings. One such process, in this case MEND, may place another, the application program, into a single step mode where the latter will be stopped before each call to EVAL and again as each call returns. The MEND process will at these points be restarted and given information about what the application process is doing. MEND stores this information in a multi-level structure it creates called an MSTACK.

Each level of an MSTACK corresponds to a level of the control stack being monitored. Each level contains the original object (actually, a pointer to it) being evaluated at that level and a new object, called the "displayed object", that will or does contain the results of evaluating each of the arguments or elements of the original. The displayed object is initially the same as the original (in a real sense, it is the original) but is systematically rebuilt as each element is evaluated and replaced by what it returns. Thus MEND can keep track of the relationship between the changing display and the unchanging program. (Figure 2 shows the various stages that the displayed object corresponding to the FORM being evaluated in Figure 1 goes through. Each stage would in fact be painted over the

previous one so that all of these stages represent only one line of the screen. The down-arrow shown in some of the stages is a place-holder that represents an object that is actually expanded on the next line of the screen.) A pointer is also kept showing which element is currently above this on the stack to be evaluated unless, of course, this is the initial element of the stack.

```
<+ .A .B .C 5>
<+ ↓ .B .C 5>
<+ 6 .B .C 5>
<+ 6 ↓ .C 5>
<+ 6 8 .C 5>
<+ 6 8 ↓ 5>
<+ 6 8 12 5>
31
```

Figure 2

III.3 A Displayed Representation of Program Execution

The printed representation of the stack occupies the top section of the screen and is the most prominent and important characteristic of MEND. Most often, in fact, it is only necessary to watch this display as the application program is executed to ascertain where a bug is located or to observe exactly how the program operates. Therefore considerable time has been spent making the operation of the MSTACK and associated display as natural and informative as possible.

Using the collected information described in the previous section, a representation of the stack is displayed as a number of lines, each corresponding to one level. Each line shows a level number and the displayed object printed in an abbreviated form (known as "&-printing", named for the printing function &[14] created by Greg Pfister). Although strictly speaking the stack builds upward (towards higher memory locations), it seems more natural to display and speak of

the stack as building from the top downward, in the direction that printing normally occurs. As each element of the bottom level (the bottom line on this section of the screen) is placed on the stack for evaluation, a new line is added beneath the previous bottom-level line showing that element, and the element is replaced in the previous line by a pointer ("↓") marking its location. When the element finally returns a value, the returned value will replace the pointer in the displayed object.

To avoid visual distraction, a minimum of updating is done on the screen. In most cases random access is used to replace only those lines that have changed. In general the complete stack will not fit in the display area allocated for it (whose size is user adjustable) so a scrolling procedure has been devised. The complete display area is rewritten whenever an attempt is made to write past the bottom or erase upward past a certain level while some lines are invisible because they have been scrolled off. The scrolling parameters have been selected to optimize the number of lines visible on the average vs. the frequency of scrolling. Level numbers allow the user to see how much is hidden "above the screen" and how deep the evaluation is nested.

III.4 MEND Program Steps Vs. MDL Steps

To make it easier for the programmer to follow what the application program is doing, the speed of execution of the program must be controllable. MEND does this by inserting a constant, user adjustable delay between program "steps." One MEND step is not precisely the same as one MDL step. Remember that a MDL step is one call to or return from EVAL. Each MSTACK level, and therefore each displayed line, will have two MDL steps associated with it. At the first step, MEND will create the level and add one line to the screen. At the second step, MEND will erase that line and put the returned value back into the previous line. For clarity MEND adds a third step between these two which actually occurs at the second MDL step. Before substituting into the previous line, the current one is first replaced by the returned value, the normal delay occurs, and then the "second" step occurs as described.

Some types of MDL objects are self-evaluating; EVAL will simply return the object it was given. Although nothing interesting has happened, two steps have occurred. In this case MEND will avoid clutter by pretending that no steps have occured. (MEND only does this with built-in types that MEND recognizes, and the programmer should recognize, as being self-evaluating. Programmer-defined types that are self-evaluating will generate the usual number of MEND steps.)

Another case of disparity between MDL and MEND steps is more complex. First some further explanations about MDL objects are needed. Generally the

interesting objects, the ones that generate MEND steps, are linear (usually list) structures containing a number of other MDL objects, as are the FORMs described earlier. Normally during evaluation of such an object the elements will be evaluated one at a time from first to last. However this sequence is not always followed by MDL. MEND cannot directly determine which element is about to be evaluated at each call to EVAL. It is only given the object to be evaluated itself and not its position in the parent structure. It is normally sufficient to do a comparison of this object with the elements of the parent, starting with the first element believed to not yet have been evaluated. Naturally, if the elements are evaluated out of order, this procedure may fail to find the desired match, because a MDL object may contain the same element in two or more positions. Thus it is possible to match the about-to-be-evaluated object to the wrong occurrence of it. Further complications arise because some functions can sometimes back up and re-evaluate their arguments.

A strong attempt was made to make MEND dependent only upon the general characteristics of MDL functions and not upon specific exceptions and idosyncracies. It was felt to be desirable to make MEND independent of both future changes to the language and programmer-defined "primitives" that would not be known to MEND. Besides, without actually simulating MDL it is not possible to always get the information MEND needs. It was felt that the "general rule" approach would take care of a sufficiently large number of cases without falling into the simulator problem.

It was determined after some experimentation, however, that MEND could not be made to work properly without some specific knowledge about several important cases in MDL. Two functions, PROG and REPEAT, allow for branching the flow of control. They are normally first-to-last functions as described above but at times control may jump backwards to re-evaluate some elements. MEND was implemented so that if it cannot locate an element in its normal search path, it will start looking again from the beginning of the structure. If it is then found, the displayed object will be reinitialized to be as if evaluation had not yet proceeded past that point. Evaluation will then continue normally.

Another phenomenon of MDL that we must discuss is what this author labels the "clause" behavior. A clause is a list of objects given to a function as a single argument. The list is not itself evaluated, but some or all of its elements are evaluated. The most common function illustrating this behavior is COND, a general purpose conditional function. COND's arguments are all lists of objects. It sequences through these lists, evaluating the first object in each, until an evaluated object returns something considered "true." Then the rest of the objects in that list are evaluated and COND returns what the last object in the list returns. MEND's normal search path only looks at top-level elements and would therefore never find the ones actually being evaluated.

This phenomenon seemed to be more widespread than the PROG/REPEAT one and could not be easily attributed to certain functions. The solution chosen here was to in all cases do a nested search in elements that looked as if they might be clauses. (The search actually goes one extra level deep to allow for certain special cases.) When a match is found in a clause, MEND will for clarity generate extra steps to make it appear to the user as if first the clause and then its appropriate element was put on the stack for evaluation. The clause will stay on the stack until some element that is not above it in the evaluation tree is evaluated. At that time the clause will be removed in an orderly manner, and the new element or clause plus element will be put onto the stack. To do this smoothly, up to six MEND steps may have to be generated for the one MDL step. (See the example in Appendix A.)

III.5 What the User Does Not See

MEND, in its display of the MSTACK, attempts to show the user everything of importance that is happening as the program is executed. However certain features of MDL cannot be captured in this sort of representation.

One such feature is the existence of compiled code. Although MDL was originally intended to be a high-level interpretive language, an assembler was written[15], producing machine code that executes in the MDL environment, to allow programmers to create "primitives" that perform functions not otherwise available in MDL. Once the assembler was written, it was natural that a compiler[16] followed. It translates normal MDL code into MDL assembly code which is then assembled. Typically MDL programs are tested interpretively and, when fully debugged, compiled in order to obtain a major increase in speed of execution.

A call to a compiled (or assembled) function usually looks to the programmer and to MEND like a call to a MDL primitive. The operation of the function is not seen except when it calls uncompiled functions. This does not present a major problem to MEND since generally only uncompiled functions are being debugged, and the compiled ones encountered are hopefully performing known functions properly.

One feature of MDL that MEND is unable to cope with is the interrupt system. The programmer may enable a large class of interrupts and assign handling functions wherever desired. Examples include interrupts for characters being typed to a certain input channel and notification that the system is about to be brought down. (MEND uses interrupts to catch single character commands and to catch errors.)

Recall that MEND monitors application program execution by placing its process into a single-stepping mode. When MDL passes control to an interrupt

handler, it temporarily causes the process to leave single-stepping mode. This is necessary partially because such a handler may be specified to run in a process other than the current one at the time of the interrupt. It unfortunately makes the handler invisible to MEND. It is therefore not currently possible to use MEND to debug interrupt handlers.

There is a whole series of side-effects, such as printing by the application program, that are not directly seen in the MSTACK representation but are made visible by various other features of MEND. These will be discussed where appropriate in later sections.

IV Issuing Commands to MEND

IV.1 Types of Commands

There is a need in debugging systems similar to MEND for two types of commands for controlling operation. One type is a set of short, immediate-action commands that the user of the debugging system may issue at any time, such as a command to completely stop all activities of the debugging system and to exit. The other type is the set of all commands not belonging to the first set. This includes commands that take arguments and those that can only be given while the application program is suspended from execution.

IV.2 Immediate Interrupt-Level Commands

In MEND's normal mode, various actions occur automatically. Most importantly, the application program is executing and the displayed representation of the MSTACK is being updated correspondingly. During this time the programmer may type ahead to the application program, but, since there is only one physical cursor that is used by both MEND's display and typein echoing that must therefore jump back and forth between the two text entry positions, a large amount of typein is awkward. Therefore, all immediate action commands that the user can issue while MEND is in automatic mode are single characters. To minimize the chance of conflict with typein that the executing application program might read, MEND immediate action commands are invoked by typing certain ASCII control characters. Furthermore typing one of these control characters causes an interrupt to occur which is handled by MEND and allows immediate response.

Note that this arrangement requires that there be certain reserved characters, as listed below, that cannot be used to communicate with the application program. Normally this presents no difficulty. An alternative method was considered that required the reservation of only one control character. However, the user would be required to type an additional character as an argument signifying what function is intended. There did not seem to be a great advantage to this scheme and it was considered after the first scheme had been implemented. For this and other reasons stated later in this section, the alternative scheme was discarded.

From the user's viewpoint, the currently implemented interrupt-level commands are as follows. (ASCII control characters are represented by "↑" followed by the corresponding letter.)

↑L	clear screen, reprint stack and input
↑Q	Quit from current MEND
↑E	End automatic mode
↑B	Begin automatic mode
↑U	Unprint (stop displaying stack)
↑P	Print (start displaying stack)
↑O	skip Over (completely evaluate) current object
↑N	Next step (used when not in automatic mode)

The command invoked by typing ↑L is for housekeeping purposes. It causes extraneous information (e.g., old program output) to be cleared from the screen and all MEND-related constantly-displayed information to be reprinted. Unread input is also reprinted.

The command invoked by typing ↑Q is used to exit from an invocation of MEND. It has the effect of stopping all actions related to monitoring the application program and allowing the program to continue normally. In this way a programmer may discard MEND and continue running the application program without the need for restarting it at the beginning.

The two commands invoked by typing ↑E and ↑B switch MEND between automatic mode, the only one described thus far, and non-automatic (command) mode. Command mode will be described in the next section.

Sometimes it is desirable for the sake of saving both computer time and the application programmer's time to turn off printing of the MSTACK. MEND will continue to monitor program execution and store the appropriate information but it does not display it. Nor does it pause between steps in the normal manner. This is mainly useful when breakpoints are present to turn on printing or switch to command mode. At that time MEND will know how it got to the current level and will be able to display the MSTACK as usual. Two commands invoked by typing ↑U and ↑P toggle the state of printing.

Whenever the programmer is satisfied that a particular call of a function is working properly or for some reason he or she is not interested in seeing the details of evaluation of some object, typing ↑O just before the object's evaluation invokes a command that causes MEND to skip over that evaluation and continue normal monitoring and display immediately after a value is returned. Unlike turning off the printing, no monitoring is performed here at all so the evaluation proceeds as fast as it would without MEND. This is actually handled by causing MDL to leave single-stepping mode during the evaluation of the object.

The last command, invoked by typing ↑N, is special in that it is ignored in automatic mode. Its actions in command mode will be described in the next section. It is an interrupt-level command mainly for convenience and compatibility with DEBUGR.

Bruce Daniels' DEBUGR is the prototype MDL multiprocessing debugging system. It provides a simple user interface to MDL's single-stepping functions that makes single-stepping through a MDL program appear similar to single-stepping through an assembly-language program with DDT[2,3]. The choices of characters for invoking many of the MEND functions were based upon the characters used to invoke similar functions in DEBUGR. Many of DEBUGR's command invocation characters were in turn based upon those in DDT. The object of all of this is to build character/command associations in the user's mind that may be carried over from one system to the next.

Several other control-character commands are handled by MEND invisibly. That is, the user may not be aware that they are being handled. Most of these are commands that are already being handled by other subsystems but must be intercepted by MEND to maintain consistency in its environment. Two of this type, invoked by typing ↑S and ↑G, are already set up in an initial MDL and a third, invoked by typing ↑F, is set up by the subsystem EDIT, which is called by MEND as described in the next section. All three of these are used to escape to various command levels. MEND has its own command level (see next section) and must in many cases escape to that one instead to maintain control. Until a method was discovered for having ITS section the screen, it was necessary to also interrupt on carriage-returns (↑M, the new-line character) on input to insure that typein was kept in the proper section.

IV.3 MEND's Command Level

Sometimes the user may wish to stop the application program at some point to examine it in more detail or to alter it or its environment in some way. Alternatively the user may want to issue commands to MEND that require arguments. A command level is provided for both of these activities.

The command level differs from the automatic mode described in the last section in two ways. First, the application program is not continually executing. It runs one step at a time under the direct control of the user rather than automatically. Second, in non-automatic mode typein is normally passed to MEND instead of the application program, even that class of typein which does not produce immediate action.

When the application program is stopped, the user may either request a service from the debugging system or examine and/or modify the program and/or its environment. Since editing functions form a large part of the latter class of actions, it was decided that instead of requiring the user to "import" an editor, MEND should provide an editor by making one available at command level.

It is generally believed and empirical evidence indicates that creating a

new editor is "the kiss of death" for a MDL subsystem. A number of MDL editors have been tried over the years and the only one that finally became generally accepted (and is now pre-loaded in MDL) is EDIT[11]. Members of the MDL community will tolerate minor changes to EDIT, but they will not accept a new editing system. The situation is analogous to one of the major obstacles blocking the acceptance of ESP. Programmers were accustomed to using DDT to examine and modify their machine-language programs. ESP was not compatible with DDT and therefore did not provide the familiar interface desired.

In view of the situation, it seemed desirable to incorporate EDIT into the command level, essentially unaltered. EDIT uses a special reader that either interprets input as a command to it or passes input on to the normal MDL reader. At first the task of superimposing a MEND reader on top of both of these appeared difficult. After much discussion with the current maintainer of EDIT, a very satisfactory solution was arrived at.

A general capability was added to EDIT allowing the specification at runtime of a table of EDIT-like commands to be handled by programmer-defined functions. This table is searched before EDIT's command table and thereby provides a capability to override standard EDIT commands. MEND basically uses an invocation of EDIT as its command level with all MEND commands included in a table as described. EDIT-format commands are all one or two letters and may take suffix arguments. Currently the MEND table includes the following commands. (FIX here means a MDL object of type FIX, i.e., a fixed-point number. FLOAT means a floating-point number. PPRINT[14] is a function that prints a MDL object in a format which indicates the positions of its elements and sub-elements in the tree hierarchy.)

NAME	ARG TYPE	MEANING
?	none	type out short summary of MEND commands
??	none	type out complete summary of all commands
O	any	Open object or MSTACK level n (if arg n is FIX)
V	none	toggle Verbosity
N	FIX	do Next n steps (like ↑N)
OV	none	skip OVer current object (like ↑O)
Q	none	Quit EDIT & return to automatic mode (like ↑B)
QM	none	Quit MEND (like ↑Q)
SN	FIX	Set Number of lines used for stack display
SV	FIX	Set leVel below which MEND will not 1STEP
SD	number	Set Delay time (FIX or FLOAT) between steps
PD	FIX	Pprint Displayed object in level n
PO	FIX	Pprint Original (actual) object in level n
AI	any	Add an Item to the list being monitored
DI	FIX	Delete Item number n from the list
PI	FIX	Pprint Item number n

What the user types to the command level is inspected by EDIT. If it looks like an EDIT command, it is looked-up in the MEND command table or the EDIT one and handled as appropriate. Otherwise it is passed to the MDL reader. In general any interesting (useful) input that MDL should read and evaluate will not look like an EDIT command, so there is no confusion.

As can be seen from the table, the MEND commands include general information retrieval ones (? and ??), system tailoring ones (V, SN, SV, and SD), and commands that allow objects to be accessed in special MEND locations (O, PD, PO, AI, DI, and PI). The O command is special in that it allows an object known only by its location in the MSTACK to be opened for examination and alteration by the normal EDIT commands. The remaining commands duplicate control-character commands except that they are read at normal input level rather than at interrupt level.

The most powerful feature of this command level is seen when MDL objects are evaluated. This level, and therefore the evaluation, uses stack space below the application program and in the same process. All bindings and pending evaluations of the program are above the command level where they may be

examined and modified at will. Most of what constitutes MEND is in another process safely removed and hidden from the user. Even the small amount of overhead constituting the command level function is hidden from functions such as FRAMES[13], which may be used to view the levels of the application program's current control stack. The effect is much the same as that of MDL's listening loop which is invoked at the current bottom of the stack in case of error, to allow the programmer to run any program (evaluate any object) to find and correct the problem.

Note that for critical examination of the program, the user may request that it be continued for a limited number of steps (usually one) with control then being returned to him or her at the command level. One normally would employ such a feature when the program is executing statements near the area of suspected trouble but it is not entirely clear how or why the program is misbehaving.

IV.4 Breakpoints

Normally MEND operates in automatic mode where the application program is running continuously. If one is aware of a particular area of concern, one cannot and should not have to see that area as it is reached and quickly type ↑E in order to stop the program. One may have the program running with a very small delay time or have the printing turned off.

In their simplest form, MEND breakpoints stop the program when evaluated, putting the user in command mode if she or he was not already in it. This is the MDL equivalent of DDT breakpoints, which stop a program at a certain instruction. The breakpoints can also be conditional, where the evaluation of an arbitrary object determines whether or not to actually break at that point.

The breakpoints as so far described are very much like the breakpoints that can be set by EDIT. As a matter of fact, EDIT is used to set and clear these breakpoints just as it would be in the absence of MEND. The only real difference is that, when MEND is present, a MEND function is called to handle the break instead of an EDIT function.

MEND breakpoints would be useful even if they only did what was just described. However they are more powerful. A standard EDIT-style breakpoint is a call to the break function with a number of arguments. The first of these is the object at which the breakpoint is set, which will be evaluated when the user returns from the breakpoint and whose value will be returned by the break function to the application program. The next is the conditional object and the rest are objects to be evaluated and printed at each break. MEND's break function handles these as EDIT's does but also looks for a recognized ATOM

(variable name), which may come after or replace the conditional. If such an ATOM is found, it will cause a special action to occur instead of stopping the program:

ON	printing will be turned on (like ↑P)
OFF	printing will be turned off (like ↑U)
GO	free-run the object (like ↑O)
PRINT	just print arguments & continue

Other arguments will still be printed and all actions are still predicated upon the value of the conditional.

The first two special ATOMs (ON and OFF), perhaps coupled with manual control, are used to allow the MSTACK to be viewed only during areas of interest while maintaining maximum speed in other places. The next one (GO) is used to avoid wasting time examining the details of a functional call that is known to be working and is probably evaluated repeatedly. When the break function returns, single-stepping resumes as before the break. The last special ATOM (PRINT) is used to give information at key places that might not otherwise be seen.

One thing that was considered extremely important was to make MEND's breakpoints compatible with those of EDIT. The problem is that a breakpoint inserted outside of MEND might still be present when MEND is started and vice versa. As has been stated EDIT breakpoints work almost exactly as usual when handled by MEND's break function. The converse is not quite true. EDIT's break function will ignore the significance of a special ATOM and simply print it as a normal argument. Fortunately, an ATOM evaluates to itself. If it is in the conditional position it will always be considered "true" and no harm will be done.

It would be easy to extend this arrangement to other special ATOMs if other functions were considered useful. The above seem to form a basic set in this system. At least they are useful and no one has yet suggested another type of breakpoint that would also prove useful.

V Final Thoughts about MEND

V.1 Monitoring of Values

A feature that was originally considered to be important, and is present in ESP[4,5], is the constant monitoring of values. This feature has not been too happily received in MEND, but that may be due more to the current implementation than to an inherent lack of utility.

Ideally one should not have to see the object being monitored or its value if one does not want to. The debugging system should be capable of performing certain actions upon, for example, a change in value. This would be analogous to conditional breakpoints except that the test would be performed after each step of execution of the application program rather than at certain arbitrary times.

One problem with the above scheme is the difficulty of testing for a change in value (the basic test). For example, the value of an object being monitored may be some sort of structure. During execution the application program may not explicitly change the value of the object but may change an element of the structure that the object evaluates to. Assuming MEND had saved a pointer to the structure, if it now evaluated the object being monitored, it would again find that structure and assume that the value had not changed. However since an element of the structure had changed, the programmer would probably want to be notified.

The only solution to the above dilemma is to at each step save a complete representation to all levels of the value of the object in some form that cannot be affected by the application program. This can be done by, at each step, unparsing the value to be saved for future tests into its printed (string) representation. Then the comparison of current and previous values will simply be a string comparison, not subject to sharing by the values. Unfortunately this may create incredible amounts of "garbage." In fact, a circular list (quite legal) will produce a string of infinite length!

The solution finally chosen was to print the values of the objects being monitored at each step and to let the user visually compare versions. The values are &-printed as usual but may be examined in full at will.

When tested it became immediately obvious that the constant reprinting wasted time and was distracting. A feature was added to cause printing to occur only at controllable intervals, but that produced other problems. Primarily the user might not see a change that had occurred at one step until some later step when valuable information had already been lost. One would also not have the opportunity to take immediate action.

What needs to be done at this point is to reduce the emphasis on such pathological cases as described above. Printing should only occur at well defined

moments (i.e., when requested or when some recognizable condition occurs). Only gross and easily testable changes in value should be watched for (i.e., the value is a new object, entirely). Conditional monitoring should be installed analogous to the conditions of breakpoints. With all of this testing, a variety of indications should be available as with breakpoints.

V.2 Control Stack Display

MEND's display of the application program's control stack has proven to be by itself a highly useful debugging aid. This display can be and often is used without any of MEND's other features to debug a faulty program. It should also be emphasized that a valid use of this display is to monitor a working program in order to understand its operation. In this case, where the user of MEND is completely unfamiliar with the operation of the application program, MEND's step-by-step display of program execution is even more useful than in the normal debugging situation.

One of the unique features of MEND's "trace" of the application program is that MEND shows both the code being executed and the results of that execution. Since MDL is an applicative language, most of the results consist of values returned by evaluated objects. By reflecting the returned values back into the original code, MEND shows intermediate results in an intuitive manner for the user.

Other programming languages could also benefit from this sort of display. Too often a trace routine provided for debugging in a language shows only results of execution, such as value assignments, without showing the corresponding program steps, or it shows program steps but leaves it to the programmer to insert instructions to show intermediate results. What the programmer really needs to see is a well integrated display of both program instructions and results, such as MEND provides.

MDL is an exceptional language in that the MEND display was implemented fairly cleanly with just a package of MDL functions and without changes to the interpreter itself. With suitable language enhancements, however, a similar display could be provided for LISP or any other LISP-like language. Although major compiler modifications would be required, a stack-oriented display might also be suitable for a block structured language like ALGOL.

Especially in a block structured language, where instructions might be treated in groups rather than individually, but also in applicative languages, it would be useful to be shown variable bindings as they occur. MDL does not provide any information about the occurrence of bindings, so the programmer is not informed of such occurrences by MEND. However when major modifications are made in a language to provide a display similar to MEND's, it would probably be a

mistake not to include a provision for notifying the programmer of variable bindings and unbindings as they occur.

V.3 Simulation Vs. Multiprocessing

Besides its control stack display, MEND's other major feature, and its major difference from debugging systems like ESP or the first version of MUMBLE, is its use of multiprocessing rather than simulation to follow application program execution. (Even debugging aids that use multiprocessing often do not work the way MEND does. While MEND is a program written in the same language and running concurrently in the same interpreter as the application program, some debuggers, such as DDT, are machine language programs running "near" the application program, and others are simply compiled into the language itself.) This feature is perhaps the key to MEND's success as a debugger. The choice to avoid simulation certainly reduced the size and complexity of MEND by two or three times and similarly affected run time. What remains to be discussed are the tradeoffs involved in the choice. The advantages just mentioned heavily favor the multiprocessing solution but there are other considerations.

If MEND had been implemented using simulation, it would for the evaluation of each object determine which elements need evaluation; evaluate those elements by recursively calling MEND's own evaluation simulator; determine whether some function should be applied; determine the effects of the required functional application, if any; and cause those effects while displaying some representation of them for the programmer. In short, MEND would duplicate the actions of MDL while maintaining and updating its own information about those actions. Therefore MEND would at all times know exactly what the application program was doing and could never miss some important effects of execution or be fooled by an unusual program. In the previous section it was stated that it is desirable to show the occurrence of variable bindings to the programmer. This feature would be easy to implement in a debugging system that simulated execution of the application program.

But what about a debugging system using multiprocessing? A multiprocessing debugger, such as MEND, allows the application program to execute more-or-less freely in its own process while monitoring it from another process. Since MEND is not in direct control of the application program, it must rely for its information on whatever data it is given by the multiprocessing mechanism built into MDL and on inferences based on examination of the application program's environment, its knowledge of the general behavior of MDL programs, and some specific knowledge of special cases. The occurrence of variable bindings is not decipherable from the information that MEND can gather.

Fortunately, as has been previously shown, MEND does have the information essential to the creation of its display and the display is a sufficient debugging tool.

Monitoring execution of the application program from another process provides sufficient information for a useful debugging system while simulation of the application program provides enough information for a more comprehensive system. Therefore, if speed and size of the debugger were not important factors, then, although not a necessary choice, simulation would be the favored choice for a debugging system except for one other factor.

The simulator needs much more knowledge of specific details of the behavior of program functions than does the monitor. In fact the simulator must know exactly the effects of each function that might be used in the application program. Uncompiled user-defined functions are no problem. They consist merely of one or more applications of other functions to given arguments. Compiled user-defined functions cannot be dealt with at all without actually creating a simulator for machine language programs, a project comparable in magnitude to the design and implementation of all of the other parts of the simulator combined! The remaining functions are MDL's built-in primitives. They are well defined but frequently change (usually upward compatibly). Furthermore new primitives are constantly being added. A simulator for MDL programs would therefore quickly become outdated.

Given the problems of a simulator and the relatively few drawbacks of a monitor, it is clear now why MEND was implemented as the latter. The same arguments apply to debugging systems for other languages. Whenever the appropriate multiprocessing features exist, with sufficient information available about execution in another process, a debugging system based upon simulation of the application program should be rejected in favor of a monitor-type system. MEND has proven to be an effective debugging aid and should serve as a good example of the latter type of system.

VI Suggestions for Future Work

VI.1 Monitoring of Access

Because of the problems described in Section V.1 concerning the monitoring of values, there was not enough time to work on a related but more interesting feature for MEND. Frequently the main piece of information that is available about a bug in a program is that at some point a certain data area (variable value, list slot, etc.) is being "clobbered" by function(s) unknown.

Rather than watching the program execution in detail to find the culprit, it would be far more useful to set a "breakpoint" on access to the location in question. This could be done by having MEND redefine all data access functions to watch for certain locations. Not only would that be messy, but it goes against one of the basic philosophies of MEND. With good reason, as described earlier, MEND tries to alter its environment as little as possible. (For example, what if the application program redefined one of the functions also?)

MDL again provides the answer. The code already exists for monitoring read or write access to any standard data location. It is only necessary to set up the proper type of interrupt handler with a pointer to the location to be watched. Each time the specified type of access occurs, the handler will be called with all of the particulars.

MEND should have commands installed for creating and destroying such handlers. For consistency and to facilitate remembering for the user, the possible actions of this handler upon being called should be similar to those of breakpoints and of monitoring of values.

VI.2 Other Unimplemented Features

Three other proposed features were not implemented due to an acquired belief that the value of each of these features in relation to the goals of this work was not worth the time required to properly implement it. However each of these features has merit and may be desirable in some future, more comprehensive debugging system.

The first such feature involves saving all output of the application program for the programmer to refer back to. This has the implementation difficulty that there is no easy way to separate application program output from much of the MEND output. All application program output and certain MEND output goes to the lower section of the screen utilizing the same output mechanism (i.e., standard MDL output to the primary terminal output channel). No distinction is made between the two kinds of output. A distinction could be made by having

MEND do all of its output through yet another terminal output channel (a second channel is currently used for the upper screen section), but another consideration made the effort seem not worthwhile. In practice, programs that produce a lot of output usually send it to a file and not to the terminal. Since only small amounts of output are usually sent to the terminal, a short output history, such as that which is present on the screen itself, is generally sufficient.

The second unimplemented feature would have allowed the setting of MEND breakpoints using the editor IMED. This would have meant sending a function out to the IMLAC where local editing would have been used to create or delete such breakpoints. PRINTTYPE and READ-TABLEs would have been used to allow for setting normal breakpoints and the special MEND types (ON, OFF, etc.) using one or two mnemonic characters.

The primary reason that this feature was not implemented was that EDIT was chosen to be the MEND editor instead of IMED. That choice was made partly because EDIT is in many respects the more powerful of the two, but mostly because EDIT is the editor now used almost exclusively by MDL programmers. In fact, EDIT is now pre-loaded in MDL while IMED is not. Another reason for rejection of this feature was that it would have made MEND, or at least this part of it, terminal dependent.

In a different environment a similar feature might be quite useful. IMED still has the large advantage over EDIT that by constantly displaying the entire function while the programmer moves the cursor around in it, creating and deleting MEND "command symbols" at will, the user is provided with a much better feel for the debugging environment being set up than with EDIT.

The third feature would allow the user to reverse execution of the application program or to simply back up to some previous point. This was suggested in two varieties, an actual undoing of all effects of the program on a step-by-step basis or simply showing previous states of the MSTACK. The first version would have used UNDO[10], a package of functions to actually back up a program to some previous state. UNDO however violates a primary design goal of MEND. It works by redefining every MDL function that has a side-effect, thereby causing most of the negative effects of simulation that were previously discussed. Unfortunately the way UNDO works is really the only reasonable way such a package could work, short of modifying MDL itself, and even UNDO is not foolproof.

The only practical way this feature could be implemented is in its second variety. It would be possible to store (in a file, probably) information specifying the state of the MSTACK at each step and to allow the programmer to view previous steps in some comfortable manner. The time required to implement this feature, though, put it outside of the scope of this work.

VI.3 Immediate Action Commands

Empirical evidence suggests that one more interrupt-level immediate action command would be highly useful. Currently the user may skip over the complete execution of a single object by typing ↑O just after the object is placed on the stack for evaluation. The user may also want to rapidly complete the evaluation of some object that is already executing. For example, one may have watched the first few objects in a REPEAT loop being evaluated and now wants to free-run the application program until the looping is finished. For such a case, a command should be available to skip over the evaluation of the current object and all others at the same level (i.e., finish evaluation of the object on the previous level).

Further use of MEND may indicate a need for other interrupt-level commands. (Perhaps certain interrupt-level commands should somehow be able to take arguments?)

VI.4 Terminal Handling

The MSC implementation of MEND was discarded because of problems specific to the current operating environment. In a suitable environment, multi-screening is still seen to be a useful feature for a similar debugging system. It has even been suggested that something of the sort could be implemented using two separate terminals. One terminal would look to the application program just like the terminal it would have seen in the absence of MEND. The other terminal would be used for communications with MEND. Not only would this eliminate output conflicts between the application program and MEND, but with two keyboards there would be no need to have a reserved character set for MEND interrupt-level commands. Of course, it might be difficult for the user of such a system to coordinate activities between the two terminals.

Another area where MEND might be improved is its knowledge of the terminal being used. As has been previously discussed, it currently assumes certain basic display functions and relies upon ITS to understand the requirements of the terminal. Although ITS handles the job well, it is not the only operating system that MDL may run under. This author, in fact, actively uses MDL under TENEX, an operating system that lacks the terminal code to support MEND. To properly work under most operating systems, MEND would have to be tailorable to individual terminal codes and requirements and would have to do much more of the work than under ITS.

It has been proposed that MEND could be made to work in some fashion with the basic ASCII printing terminal, something that nearly all terminals and

operating systems can simulate. However it is the opinion of this author that MEND would lose much of its value under such conditions. Watching the stack grow and shrink, and seeing objects replaced by their values, gives the user more of a "feel" for what the application program is doing than a long stream of sequentially printed lines ever could.

Appendix A: Sample Display

What follows is an excerpt from a sample MEND debugging session showing the main display area (at the top of the screen) for a series of consecutive steps. That which appears between the dotted lines is what was displayed at each step of the program. The program being so displayed is a simple exponentiation function being applied to 2 and 3. The definition of this function is printed in full (using PPRINT) before the sample displays.

The words printed on the separator line show the state of several flags. "Print," "auto," and "next" mean that printing is enabled, MEND is in automatic run mode, and MEND is waiting for the next object to be typed, respectively. The numbers on the left show the depth of evaluation. The initial object, as originally typed by the programmer, is placed at level 0. "MENDING" was returned by MEND when the system was started. The next line shows the user's typein of the initial object (terminated by "$"). Below this line, in the last step, MDL printed the object that the initial object returned. Explanatory comments appear to the right of semicolons.

Here is the definition of the exponentiation function:

```
<DEFINE EXP (X Y)
        <COND (<0? .Y> 1)
              (<* <EXP .X <- .Y 1>> .X>)>>
```

```
--------------------
 . . .                          ; Several initial displays omitted.
--------------------

0       <EXP 2 3>               ; Initial object being evaluated.
1       <COND (#FALSE () 1) ϕ>  ; Body of the EXP function is just a COND FORM.
2       (ϕ)                     ; Clause of the previous COND being evaluated.
3       <* ϕ .X>                ; Only element of the above clause.
4       <EXP 2 2>
5       <COND (#FALSE () 1) ϕ>
6       (ϕ)
7       <* ϕ .X>
8       <EXP 2 ϕ>
9       <- .Y 1>                ; Most current element being evaluated.

                                                              print___auto__
"MENDING"                       ; Message returned by function that starts MEND.
<EXP 2 3>$                      ; Object to be evaluated and debugged.

--------------------
```

```
--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) !>
2       (!)
3       <* ! .X>
4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 !>
9       <- ! 1>
10      .Y

_____________________________________________________________print___auto__
"MENDING"
<EXP 2 3>$

--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) !>
2       (!)
3       <* ! .X>
4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 !>
9       <- ! 1>
10      2

_____________________________________________________________print___auto__
"MENDING"
<EXP 2 3>$

--------------------
```

```
--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) !>
2       (!)
3       <* ! .X>
4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 !>
9       <- 2 1>


_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) !>
2       (!)
3       <* ! .X>
4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 !>
9       1


_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------
```

```
------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) !>
2       (!)
3       <* ! .X>
4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 1>


_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) !>
2       (!)
3       <* ! .X>
4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 1>
9       <COND (<0? .Y> 1) (<* <EXP .X <- .Y 1>> .X>)>


_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------
```

```
--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) !>
2       (!)
3       <* ! .X>
4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 1>
9       <COND ! (<* <EXP .X <- .Y 1>> .X>)>
10      (<0? .Y> 1)

_____________________________________________________________________print___auto__
"MENDING"
<EXP 2 3>$

--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) !>
2       (!)
3       <* ! .X>
4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 1>
9       <COND ! (<* <EXP .X <- .Y 1>> .X>)>
10      (! 1)
11      <0? .Y>
_____________________________________________________________________print___auto__
"MENDING"
<EXP 2 3>$

--------------------
```

```
--------------------

4       <EXP 2 2>                ; Scrolling has occurred since the last display
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 1>
9       <COND ! (<* <EXP .X <- .Y 1>> .X>)>
10      (! 1)
11      <0? !>
12      .Y


_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------

4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 1>
9       <COND ! (<* <EXP .X <- .Y 1>> .X>)>
10      (! 1)
11      <0? !>
12      1


_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------
```

```
--------------------

4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 1>
9       <COND ! (<* <EXP .X <- .Y 1>> .X>)>
10      (! 1)
11      <0? 1>
```

```
                                                                print___auto__
"MENDING"
<EXP 2 3>$

--------------------

4       <EXP 2 2>
5       <COND (#FALSE () 1) !>
6       (!)
7       <* ! .X>
8       <EXP 2 1>
9       <COND ! (<* <EXP .X <- .Y 1>> .X>)>
10      (! 1)
11      #FALSE ()
```

```
                                                                print___auto__
"MENDING"
<EXP 2 3>$

--------------------
```

```
-------------------
 . . .                          ; Several displays omitted.
-------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) ϕ>
2       (ϕ)
3       <* 4 2>
```

```
_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) ϕ>
2       (ϕ)
3       8
```

```
_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------
```

```
--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) ✝>
2       (8)
```

```
_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------

0       <EXP 2 3>
1       <COND (#FALSE () 1) (8)>
```

```
_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------
```

```
-------------------

0       <EXP 2 3>
1       8
```

```
_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------

0       <EXP 2 3>
```

```
_____________________________________________________________print____auto__
"MENDING"
<EXP 2 3>$

--------------------
```

```
--------------------

0       8                       ; Initial object returns this value.
```

```
_____________________________________________________________print___auto__
"MENDING"
<EXP 2 3>$

--------------------

0       8
```

```
_____________________________________________________next___print___auto__
"MENDING"
<EXP 2 3>$
8                       ; Final value is returned by MDL in correct screen location.

---------------------
```

Appendix B: Glossary of Terms

&[14] A function pre-loaded in MDL that prints objects in an abbreviated form to fit in a programmer-specified number of character positions.

1STEP[8] A built-in MDL function used by one program to single-step another for debugging purposes.

ATOM[8] A MDL variable.

COND[8] A built-in MDL function providing a general conditional capability. The arguments to COND are lists. MDL evaluates the first element of each list in turn until an element returns a non-false value. Then the rest of the elements in the current list are evaluated and COND returns what the last element in the list returns.

DDT[2,3] A program used to debug assembly language programs.

DEBUGR A program utilizing MDL's single-stepping functions to show a programmer the step-by-step execution of a program. DEBUGR is an attempt to provide a DDT-like debugger for MDL programs.

EDIT[11] A pre-loaded editor for MDL objects. EDIT works within MDL by restructuring the object being EDITed according to the specifications of the programmer. EDIT allows one to define one's own commands or to redefine those of EDIT.

ENVIRONMENT[8] A MDL object which specifies a particular set of variable bindings. An ENVIRONMENT is normally cumulatively built up as the control stack of a program builds and ATOMs are bound. The ENVIRONMENT actually corresponds to a particular point on the control stack. A program may have an object evaluated in an ENVIRONMENT specifying the current state of another running program. The effect is as if the latter program had evaluated the object.

ESP[4,5] A debugging system for assembly-language programs.

EVAL[8] A built-in MDL function that evaluates an object and returns the

value of that object.

FIX[8] A MDL object that is an integer.

FLOAT[8] A MDL object that is a floating-point number.

FORM[8] A list of MDL objects which is evaluated by applying the first element (some function) to the rest of the elements (its arguments). "Execution" in MDL generally refers to the evaluation of a FORM. The evaluation of that FORM will often require the evaluation of other FORMs (the arguments to the function or FORMs in the body of the function).

FRAMES[14] A pre-loaded function that shows the programmer a printed representation of the control stack of the current program.

IMED An editor for MDL objects that works by outputting an object to the IMLAC where local editing functions are used.

IMLAC A minicomputer with a keyboard and CRT display used as an intelligent terminal.

ITS[12] A general-purpose time-sharing operating system developed by the Artificial Intelligence Laboratory at M.I.T.

LVAL[8] A built-in MDL function that returns the local value of a given ATOM. The local value is that which was last bound to the ATOM in the current ENVIRONMENT.

MDL[8] An applicative programming language used to implement MEND.

MEND Mdl Executor, aNalyzer, and Debugger. The subject of this report.

MSC Multi-Screen Console program for an IMLAC. This gives the IMLAC used as a terminal a capability for having several virtual screens that may be accessed and displayed independently.

MSTACK A structure that MEND builds to contain a representation of the control stack of the application program.

[13] P.D. Lebling, SSV User's Manual, SYS.52.07, Programming Technology Division Document, Laboratory for Computer Science, M.I.T., (in preparation)

[14] S.W. Galley, Pre-loaded Pure MDL RSUBRs, SYS.11.28, Programming Technology Division Document, Laboratory for Computer Science, M.I.T., November, 1975

[15] B. Daniels, The MDL Assembler, SYS.11.07, Programming Technology Division Document, Laboratory for Computer Science, M.I.T., (in preparation)

[16] C. Reeve, The MDL Compiler, SYS.11.25, Programming Technology Division Document, Laboratory for Computer Science, M.I.T., (in preparation)