Massachusetts Institute of Technology
Project MAC

CC-247

Memorandum MAC-M-205
November 25, 1964

MADBUG: A MAD DEBUGGING SYSTEM

by Robert S. Fabry

MADBUG is a system under which the user can create and debug programs written in the MAD programming language. MADBUG allows the user to input and edit symbolic programs and to execute in a controlled way and interrogate the derived machine language programs. The most important consideration in the design of MADBUG was ease in learning and using, both for the beginner and for the advanced programmer. MADBUG is unusual in that it utilizes information which has been previously ignored. This information comes from: (1) the sequence in which the user types his requests, (2) the files available in the user's file directory, (3) the expanded information content of the new MAD symbol table files developed for MADBUG, and (4) the information inherent in the very limited, stylized set of coding sequences generated by a compiler. The use of this additional information manifests itself in two ways: (1) the user need provide very little information to accomplish a given task, and (2) the user does not have to understand assembly languages, machine languages, octal numbers,

* Early research on the form of the MADBUG system was sponsored by the Advanced Research Projects Agency under contract number SD-162 with Information International Inc. of Cambridge, Mass; later research and programming was carried out as a special project for credit under Professor F. J. Corbato; and the remaining work was carried out with no sponsorship.

relative or absolute addresses, symbol tables, machine representations of constants, or any of a host of similar items. The MADBUG requests of CHANGE, DELETE, INSERT, and APPEND demonstrate the influence of the "Expensive Typewriter" program written for the PDP-1 by Steve Piner. The "DDT" program written for the PDP-1 by Robert Saunders and the "FLIT" program written for the TX-0 by Jack Dennis and Thomas Stockham have influenced the OPEN, VERIFY, BREAK, and KILL requests.

AN INTRODUCTION TO MADBUG

A simple hypothetical session with MADBUG will provide a useful background for the detailed description which follows. Consider a user who is writing a function which returns the smallest factor of the number given to it as an argument. The function returns zero if the argument is a prime. (Lines will be prefixed with a U for the user, M for MADBUG, S for CTSS, or P for the user's program. Typing is assumed to be on a 1050 console, which means that the user types in lower case and the machine types in upper case. "M:", by itself, means a blank line typed by M.) First, the user will tell the system he wishes to use MADBUG:

```
U:madbug
S:W 1412.2
M:
```

MADBUG is now waiting for the user to give his first request. The user tells MADBUG that he is working on a subprogram in a file called FACTOR MAD:

```
U:work factor
```

MADBUG makes no response unless the request calls for information. The user wants to input the program, which doesn't exist yet. He chooses to request MADBUG to append some input to the (now empty) symbolic program (introducing, for the demonstration, a syntactic error in the third card and a bug in the initialization of the index "i"):

```
U:append
U:              external function (arg)
U:              normal mode is integer
U:              entry to factor
U:              through loop, for i=1,1,i*i.g.arg
U:loop          whenever (arg/i)*i.e.arg,function return i
U:              function return 0
U:              end of function
U:
```

The blank line signals the end of inputing cards and implies that the user will type a request next. The user requests that his program be translated into machine language:

```
U:translate
M:TRANSLATING FACTOR...
M: ***** ERROR 17025 IN STATEMENT BEGINNING ON CARD 003
M:       ILLEGAL FORMATION OR PUNCTUATION.
M:       TRANSLATION ERROR
M:
```

He asks MADBUG to print the offending third card:

```
U:print 3
M:              ENTRY TO FACTOR
M:
```

and recognizes that he omitted the period. He corrects his error:

```
U:change 3
U:              entry to factor.
U:
```

and re-translates his program, abbreviating the request name by its first letter, as is always allowed:

```
U:t
M:TRANSLATING FACTOR...
M:SUCCESSFUL.
M:
```

FACTOR is syntactically correct and the user turns to write a short main subprogram to test its operation:

```
U:work test
U:append
U:              normal mode is integer
U:loop          print comment $type.$
U:              read data
U:              fac=factor.(num)
U:              print results num,fac
U:              transfer to loop
U:              end of program
U:
```

The user is ready to test his programs. He tells MADBUG what programs to load. This does not cause loading, however. He then asks MADBUG to start his program. Since he does not specify a starting point, and since the program has not been loaded and run before, the program will be started at the beginning of the main subprogram. MADBUG will know that TEST must be translated and that loading must occur at this point:

```
U:use test factor
U:go
M:TRANSLATING TEST...
M:SUCCESSFUL.
M:LOADING FRESH CORE IMAGE...
M:SUCCESSFUL.
M:USER IN CONTROL.
P:TYPE.
```

The user is now talking to his program:

```
U:num=25 *
P:              NUM =          25,          FAC =           1
P:TYPE.
U:num=7 *
P:              NUM =           7,          FAC =           1
```

```
P:TYPE.
```

The user realizes his program has a bug, studies his program, and discovers that he should have initialized the index to 2. He could simply edit the correction into his program and GO again, but an alternate stratagy will also allow the user to discover additional bugs in his program, if any, without requiring re-translation and re-loading. The user interrupts his program and returns control to MADBUG:

```
U:(the user hits the break button once.)
S:INT. 0
M:MADBUG IN CONTROL.
M:INTERUPT WHILE IN PROGRAM WRFLXA AT 10 RELATIVE OCTAL.
M:
```

WRFLXA is one of the lowest level subroutines for console input and output. In this case, the user's program was hung there waiting for input. The user inserts a breakpoint at statement LOOP and returns control to his program. Since he does not specify a starting point, and since his program has been executed after the previous loading, the program will be restarted where it left off. The user then types another value of NUM to his program:

```
U:work factor
U:break loop
U:go
M:USER IN CONTROL.
U:num= 5 *
M:MADBUG IN CONTROL.
M:BREAKPOINT ENCOUNTERED AT STATEMENT LOOP.
```

The user fixes the initialization of the index by hand, using the open request, removes the breakpoint, and lets his program calculate the factor:

```
U:open I
M:I= 1 (U:)2
U:kill loop
U:go
M:USER IN CONTROL.
P:          NUM =          5,          FAC =          0
P:TYPE.
```

and then repeats the process for the other case:

```
U:(the user hits the break button once.)
S:INT. 0
M:MADBUG IN CONTROL.
M:INTERUPT WHILE IN PROGRAM WRFLXA AT 10 RELATIVE OCTAL.
M:
U:break loop
U:go
M:USER IN CONTROL.
U:num=25 *
M:MADBUG IN CONTROL.
M:BREAKPOINT ENCOUNTERED AT STATEMENT LOOP.
U:open I
M:I= 1 (U:)2
U:kill loop
U:go
M:USER IN CONTROL.
P:          NUM =          25,          FAC =          5
P:TYPE.
```

The user is satisfied that there are no more bugs. He does now what a less conservative user would have done as soon as the bug in the initialization of the index had been discovered. He edits the correction into the symbolic program using the MANIPULATE request. This request will replace all occurances of the first string, "I=1", by the second string, "I=2". MADBUG will list the cards on which the replacement is performed. (Editing a symbolic program whose machine language translation was used for loading will destroy the current user core image. Thus the user could not have made the change and then continued to look for additional bugs in the old core image.)

```
U:(the user hits the break button once.)
S:INT. 0
M:MADBUG IN CONTROL.
M:INTERRUPT WHILE IN PROGRAM WRFLXA AT 10 RELATIVE OCTAL.
M:
U:manipulate /i=1/i=2/
M:LOOP-1
M:THAT'S ALL.
M:
```

The user, being conservative, makes a test of his program in its final form:

```
U:go
M:TRANSLATING FACTOR...
M:SUCCESSFUL.
M:LOADING FRESH CORE IMAGE...
M:SUCCESSFUL.
M:USER IN CONTROL.
P:TYPE.
U:num=3 *
P:          NUM =          3,          FAC =          0
P:TYPE.
U:num=125 *
P:          NUM =        125,          FAC =          5
P:TYPE.
U:(the user hits the break button once.)
S:INT. 0
M:MADBUG IN CONTROL.
M:INTERRUPT WHILE IN PROGRAM WRFLXA AT 10 RELATIVE OCTAL.
M:
```

Satisfied, the user deletes the main program written for testing, and requests MADBUG to return him to CTSS:

```
U:work test
U:delete
U:quit
S:R 93.556+79.350
S:
```

CTSS is listening to the user, and the user may LOGOUT or issue any other CTSS command.

A DESCRIPTION OF MADBUG

MADBUG is instructed by requests, typed one per line. A request line is made up of the name of the request followed by its arguments, with one or more blanks for

separation. Request names may be abbreviated by their first letter. In request lines, tabulation characters are equivalent to blanks. There may be blanks before the request name and after the last argument; blank request lines are ignored. Since blanks are used as delimiters, the arguments, which may be as complicated as "a(1)+1...b-3", must be typed without internal blanks. A request which operates on variables will operate on single variables or on blocks of variables, specified in the usual MAD manner as "alpha...beta"; a request which operates on cards will operate on single cards or on blocks of cards. For example, "verify alpha beta(1)...beta(3) k(1,1,1)" would verify, in a sense described later, the variables ALPHA, BETA(1), BETA(2), BETA(3), and K(1,1,1).

MADBUG requests can be classified into four groups: the edit requests which are PRINT, DELETE, INSERT, CHANGE, APPEND, MANIPULATE, and TRANSLATE; the core requests which are GO, OPEN, VERIFY, LINKAGE, BREAK, KILL, SAVE, and RESTORE; the requests for returning to CTSS which are QUIT and EXECUTE; and the declarations which are WORK, USE, and FORCE. These requests will be discussed in the next few sections.

The Work Request

The MADBUG requests are carried out in the context of a single MAD subprogram. The WORK request allows the user to declare which subprogram is of interest. For example: "work prog" sets up MADBUG to work on the program in file PROG

MAD. The file PROG MAD does not have to exist. As illustrated in the sample session, if the user adds lines to a non-existant file, MADBUG will create the file. Thus, if the user is working in the context of a subprogram PROG, and wishes to print a subprogram ROOT, he must first request "work root" and then may request "print".

Edit Requests

MADBUG uses a different technique for editing than the CTSS EDIT command. Neither the user nor MADBUG supplies a line number for a card image. Instead of indicating a card image by giving its associated line number, the user has three options: (1) the statement label on the card, if any; (2) the card's position relative to another card which has a statement label (the third card before ALPHA is ALPHA-3; and (3) the number of the card in the deck (the 17th card in the deck is simply 17). In counting for (2) or (3), the user must count all physical card images including remark and continuation cards. MADBUG interprets the arguments of a request before executing the request; thus, if a deck consisted of three cards, "delete 1 2" would leave the third card, but "delete 1" followed on another line by "delete 2" would leave the second card.

In unusual situations there may be a long section of program with no statement labels. The user is free to insert remark cards with statement labels in such a case. MADBUG, but not the MAD translator, will allow references to statement labels on remark cards.

Three special conventions exist for specifying statement labels: (1) the "*" is always taken to mean the previous card referred to by the user, so that a "print *+3" after a "print 6" would print the 9th card, and so that a "print alpha...*+2" would print three cards starting with ALPHA. (2) the "/" is always taken to mean the last card in the deck, so that, in a five card program, "print 1 3 5" is identical to "print 1 3 /". (3) Requests which operate on cards will operate on every card in the subprogram if no cards are specified, so that "print" is identical to "print 1.../".

MADBUG observes the standard conventions of horizontal spacing: the characters after a tab will be moved to column 12 and the characters after a tab-backspace will be moved to column 11.

The description of several of the editing requests will refer to input line blocks. An input line block consists of all the lines the user types before typing a blank line. The editing requests are defined as follows:

PRINT will print all cards mentioned as arguments. Thus, "print a(1)+1...b-3" would print a block of cards starting with the card after the card labeled A(1) and ending with the third card before the card labeled B; "print 1 6 b...*+1" would print the first and sixth cards and a block of two cards starting at the card labeled B; "print" would print all of the subprogram being worked.

DELETE will delete all cards mentioned as arguments. Thus, "delete" would delete all of the cards of the subprogram being worked, and "delete 1 3...6" would delete the first and the third through sixth cards.

INSERT will insert successive input line blocks before successive cards mentioned as arguments. Thus, one might see the following sequence:

```
U:print
M:ONE
M:
U:insert
U:zero
U:
U:print
M:ZERO
M:ONE
M:
U:insert 1 one
U:a
U:
U:b
U:
U:print
M:A
M:ZERO
M:B
M:ONE
M:
```

CHANGE will replace successive cards or blocks of cards, given as arguments, by successive input line blocks. A block containing any number of cards may be replaced by an input line block of any length. Thus one might see the following sequence:

```
U:p
M:ONE
M:TWO
M:THREE
M:FOUR
```

```
M:FIVE
M:
U:c one three...five
U:a
U:b
U:
U:c
U:
U:p
M:A
M:B
M:TWO
M:C
M:
```

APPEND with no arguments will append the input line block which follows the request line to the subprogram being worked. On the other hand, if the request has arguments, they are taken to refer to MAD subprograms which will be appended, in order, to the program being worked. The following sequence illustrates using APPEND to finish writing a "program" and to re-arrange the "program":

```
U:work test
U:print
M:ONE
M:TWO
M:
U:append
U:three
U:
U:print
M:ONE
M:TWO
M:THREE
M:
U:append test test
U:delete 1...2 4 6 8...9
U:print
M:THREE
M:TWO
M:ONE
M:
```

APPEND is also useful for creating a modified version of a subprogram while keeping the original. To do this, WORK the new name, APPEND the old name, and then make modifications.

MANIPULATE is a request for character manipulation within a card image. The first argument specifies the manipulation. Arguments after the first specify cards within which the manipulation will be performed. The first argument has the form: /***/***/ where the slash stands for any separation or delimiter character which must occur exactly three times, and the strings of asterisks stand for any pair of character strings. The manipulation consists of replacing all occurances of the first string by the second string. Any character except a tab or space may be used as the delimiter; it is recognized by its being the first character of the argument. The two character strings may include any characters except the delimiter and the carriage return, and they may be of different lengths. If the first string is empty, it will be taken to match a null string before column one on the card, thus allowing a simple way of inserting a statement label on a card. As a confirmation to the user, MADBUG will print a list of cards on which the manipulation is performed. If the manipulation is performed more than once on a card, the card will be included in the list once for each time the manipulation occurs. MADBUG does not consider

replacing a string by itself to change the symbolic program. Thus the user can replace a string by itself to locate all occurences of the string. One might observe the following sequence:

```
U:p one
M:ONE          THROUGH ONE, FOR I=1,1,I.G.N.OR.X(I).E.0
M:
U:manipulate *one*loop* one
M:ONE
M:ONE
M:THAT'S ALL.
M:
U:p *
M:LOOP         THROUGH LOOP, FOR I=1,1,I.G.N.OR.X(I).E.0
M:
U:p loop+1
M:             DATA=Y(I)
M:
U:m $$label$ *
M:LOOP+1
M:THAT'S ALL.
M:
U:p *
M:LABEL        DATA=Y(I)
M:
U: m /I/J/ loop...label
M:LOOP
M:LOOP
M:LOOP
M:LABEL
M:THAT'S ALL.
M:
U:p loop...label
M:LOOP         THROUGH LOOP, FOR J=1,1,J.G.N.OR.X(J).E.0
M:LABEL        DATA=Y(J)
M:
```

TRANSLATE has no arguments, and causes the subprogram being worked to be translated into machine language by the MAD compiler. From the user's point of view MADBUG is performing the translation. It is not necessary to translate any subprogram before using it. MADBUG will request any translations that are needed at load time. The TRANSLATE request is a convenience to the user who

is changing several subprograms at one time, and who would like to catch any syntactic errors in one before turning his thoughts to another.

The Use Request

The core requests, which will be discussed in the next section, operate in the context of a core image. MADBUG must have some way of knowing what subprograms to load when creating a core image. The arguments of the USE request are the subprograms to be used. Thus a user writing a subroutine ROOT and a test program MAIN might "use main root". There are provisions for using FAP programs, special libraries, and special loader parameters; these provisions are described later.

Core Image Requests

Some core requests require cards for arguments, and their arguments observe the same conventions as those of the edit requests. A core request which refers to a declaration or remark card will operate on the first executable statement following the referenced card. Other core requests require variables for arguments. A variable is given as an argument in standard MAD notation, including multi-dimensional arrays and COMMON and ERASABLE variables, but not the dummy arguments of functions. Three special conventions exist for variables: (1) the "*" is always taken to mean the previous variable refered to by the user; (2) if no variables are specified, the request will operate on every variable in the program; and (3) the block notation

can be used to include several arrays or variables at once. Variables are taken to be ordered alphabetically (with a blank coming after R, alas.) and then by linear subscript.

The first time the user gives a core request, a core image must be created by MADBUG. This is accomplished by translating each of the needed subprograms into machine language, if necessary, loading the subprograms into core, and finally modifying some of the subprograms in order to intercept illegal references to an array. If an error is detected in this process, the core image will not be formed, and the core request will be terminated. The user should correct the error and try the core request again. The core image will be destroyed when the user issues the quit request or edits a program occuring in the core image. The core requests are defined as follows:

GO will start the user program. A single card given as an argument for GO will cause the user program to be started at the named card. If no argument is given, the user program will be started wherever it stopped last. A fresh core image will start at the beginning of the main program.

The user program will remain in control until (1) it terminates by calling DEAD, DORMNT, ENDJOB, ERROR, or EXIT; (EXIT can be implicitly called by letting control reach an END OF PROGRAM or END OF FUNCTION card.) (2) a "breakpoint" is encountered by the user program; (3) the user interrupts by pushing the break

button once; or (4) an array is referenced with subscripts pointing outside of the dimensioned array. (Some array dimension violations are not caught; this is discussed in a later section.) On any of these occasions, control returns to MADBUG, and the user is informed of the reason.

Infrequently, the user program may have an error which causes control to return to CTSS. In this case, the user should type two CTSS commands, first "save (user)" to save his own core image and second "resume (mdbg)" to return control to the core image on which MADBUG saved itself. Even if the first of these commands results in an error comment from CTSS, the user should type the second. This proceedure is called a manual restart.

OPEN will print the contents of variables mentioned as arguments, one by one, and after each, wait for the user to type a new value for the variable. If the user wishes the old value to remain, he just types a carriage return. In typing out the value of a variable, MADBUG makes use of the declared mode of the variable and of the current value to decide whether the value should be presented to the user in integer, alphabetic, floating-point, Boolean, statement label, or function mode. The user must type a constant for the new values in a form compatible with the declared mode of the variable. It is possible to change the

input/output form associated with a declared mode permanently or to override the normal associations for a single request. This is discussed later.

One special note: because of the way the MAD compiler works, one may change the effect of a transfer statement by changing the value the variable which has the same name as the statement label to which the statement transfers. One may not, however, change the scope of a THROUGH loop in this fashion, even by changing the value of the variable with the same name as the THROUGH scope.

VERIFY will cause the values of variables mentioned as arguments to be compared with the values of the same variables in a fresh, unexecuted version of core. Each variable whose value has changed will be printed with its present value. Its value in the fresh version of core will also be printed if it is non-zero.

An option is available with verify; the user may specify any core image saved with the SAVE request to be used instead of the fresh copy of core discussed above. This is done by giving the name of the saved image following the request name and before the list of variables to be varified. As the user will discover below, this name must begin with an asterisk, and can thus be recognized by MADBUG.

The discussion of output forms used for the values of variables, which was given under the OPEN request,

also holds for the VERIFY request.

LINKAGE causes MADBUG to tell the user which statement made the most recent call to the external function subprogram currently being worked.

BREAK will modify the machine language program in the current user core image so that control will return to MADBUG if one of the cards given as arguments is to be executed. When MADBUG regains control from the user program, the name of the statement which is about to be executed will be printed for the user. At this time the user will usually examine variables in his program to determine what his program is doing. "Breakpoints", as these points in the user core are called, belong to a given core image, and can vary from one saved core image to another. (See the SAVE request.)

KILL will remove any breakpoints at cards mentioned as arguments. It is not an error to insert a breakpoint where one already exists nor to remove one which does not exist. For example, to kill all the breakpoints in the subprogram being worked, "kill".

SAVE has a single name as its argument and causes a copy of the current user core image to be saved as a CTSS file with the primary name given as an argument and the secondary name SAVED. The name given by the user must begin with an asterisk. The current user

core image was produced by loading, and has been modified by execution and by MADBUG requests. One may save the current core image under a name which has already been used for a save request. In this case, the current core image will replace the previous core image. All the core images saved using the SAVE request will be destroyed when the user's current core image is destroyed. This is because the saved files created by MADBUG are not normal CTSS saved files, and are useless out of the context of MADBUG.

RESTORE will replace the current user core image with a copy of the image whose name is given as an argument. The core image name must be a name under which the user has saved a core image using the SAVE request, or it must be *FRESH. *FRESH is a byproduct of the loading process. It is a completely unexecuted version of core with no breakpoints and with all variables at their initial values. Except for the special way in which it is created, *FRESH is like any normal core image saved by the SAVE request.

Getting Back to CTSS

When the user is finished with MADBUG, and desires to return to CTSS, he should use the QUIT request. The QUIT request will destroy all the files created during the session, except for the modified MAD programs and their associated BSS and SYMTAB files.

The EXECUTE request allows the user to return to CTSS for a single command, without ending his session with MADBUG. For example, the user could effect the CTSS command "listf aa mad" by requesting "execute listf aa mad". These commands are executed using the command chaining technique with the sequence: "save (mdbg)", the user's command, and "resume (mdbg)". No provision is made for saving a core image which might result from the user's command.

SPECIALIZED FEATURES AND TECHNIQUES

Two error comments that the user may get from MADBUG have special significance. One is "TRY AGAIN.", which always means that the current request has been terminated. The other is "CONSULT LISTINGS." which can only occur as a result of a bug in MADBUG. Any user getting this comment will please retain as much information in the way of output, files, etc. as he can and call Bob Fabry, x2524, so the bug can be removed promptly. The user can often continue with more requests in spite of a "CONSULT LISTINGS." error.

Two types of improper array references are not caught. First, references with a constant linear subscript are not checked. For example, one might DIMENSION A(10) and A(20)=100. Second, references to arrays which are given as arguments to functions are not checked. For example, one could have called for ROOT.(A(K)) where K is 20. This situation can sometimes be avoided by placing arrays in COMMON, and not passing them back and forth as arguments.

In unusual cases, the user core image may "blow-up" in such a way that the information about control and about the values of variables is gone or meaningless. In this case the user will still find MADBUG a useful tool, and may approach the problem by an exponential search through time for the point at which the blow-up occurs. Stated another way, this amounts to performing a series of tests in which each test is designed to cut by a half the uncertainty about when the blow-up occurs. When the user knows the exact point of the blow-up, he can then step through very cautiously, looking for clues. Such an approach relies heavily on BREAK, KILL, SAVE and RESTORE. At the start, the user moves a core image as close to the blow-up as he knows he can, SAVEs the core image, and guesses the half-way mark, in terms of opportunities for bugs, to a place by which the blow-up must have occured. He then uses BREAK and KILL to step his current core image to the half-way point he guessed. (1) If the core image blows-up in this process, he guesses a new half-way point, half way between his saved image and his old half-way mark, RESTOREs his saved core image, and trys his new guess. (2) If the core image doesn't blow-up in the process, he SAVEs his current core image for a new starting point, guesses a new half-way mark between his new core image and the blow-up, and trys this new guess. This process is fairly simple to carry out using MADBUG, and most blow-ups can be readily solved this way.

When loading is performed, MADBUG will normally load a program named (MDBG), which MADBUG provides, immediately following the files specified by the USE request. Then MADBUG will process the core images of all programs loaded into core before (MDBG) and insert patches, using an area reserved in (MDBG), to attempt to catch any user subprogram when it accesses an array with an illegal subscript. If the user wishes to load programs which were written in FAP, MAD programs for which the symbolic programs are not available, debugged MAD programs which he does not wish to protect, or library files, he may specify the position of (MDBG) by typing (MDBG) in place of a file name in the USE request. All the files before this parameter will be treated normally, and all things after it will be ignored by MADBUG and just passed on to the loader. Any loader parameters, such as (CFLP) or (LIBE), can also be used after (MDBG). If the user needs more than eighty characters for his USE request, he may type a hypen as an argument of use. When the hyphen is encountered, MADBUG will immediately read the next input line for more arguments for the USE request. This may be done for several successive lines.

The FORCE request forces certain internal registers in MADBUG to new values, picked by the user. To FORCE a parameter, give the name of the parameter as the first argument of FORCE, and give remaining arguments as required by the parameter being forced:

FORCE PATCH will set the amount of patch space available in the user core images to the decimal number given as the argument. Initially PATCH is set to 500. The patch space is used during loading and whenever breakpoints are inserted. FORCE PATCH does not change the available patch space immediately, since the internal register is examined only during loading. A user would reduce the patch space if he was squeezed for core space. He would increase it if MADBUG complains, during loading, that there is not enough patch space, or if he exhausted the patch space inserting breakpoints. If the patch space is exhausted by breakpoints, however, it is usually sufficient to KILL some of the less neccessary breakpoints to get space for new ones.

FORCE FORMAT will set the normal input/output form associated with each of the possible modes for variables. After the word FORMAT, the arguments are taken in pairs, the first item of the pair indicates a mode and the second indicates a form. The modes are indicated by a digit from 0 to 7, standing for floating-point, integer, Boolean, function, statement label, mode 5, mode 6, and mode 7, in that order. The form designation is one of the following: "Gn" for floating point with n significant figures on output, "I" for integer, "A" for alphabetic, "P" for either integer or alphabetic with MADBUG picking for output,

"Ø" for octal, "B" for Boolean, "S" for statement label, and "F" for function. Initially, FORMAT is set to: 0 G3 1 P 2 B 3 F 4 S 5 Ø 6 Ø 7 Ø. (In this section, "Ø" is used to denote the letter "O".)

FORCE MODE allows the user to predetermine whether MADBUG saves itself as a permanant mode file or as a temporary mode file. The values of MODE are, correspondingly, "P" and "T". Mode is originally set to "P". The user will want to FORCE MODE to temporary if he is not interested in extreme reliability as much as in conserving his track allotment.

It is also possible to override all the normal I/O forms for the duration of one OPEN or VERIFY request. To do this, use one of the form designations listed above, but preceded by a slash. Insert it after VERIFY (and the saved file name, if present) or OPEN and before the arguments. For example, "open /o alpha".

MADBUG observes the convention that the first statement of a main program starts after the call to .SETUP which the compiler always inserts as the first executable machine instruction. Another convention at this level is imposed by the compiler. A breakpoint on an ENTRY TO statement will not be encountered when the entry is called, but will be encountered if control is transfered to the statement or falls to the statement.

MADBUG creates and destroys special files as it processes the user's requests. They are destroyed during the processing of the same request for which they are created. Normally, the user will not have to worry about them, but occasionally he may be made aware of their existance. (MDBG) SAVED is the name under which MADBUG saves itself when it chains to other commands. This file will vary in length during a session, but will be on the order of 30 tracks long. Its mode depends on the value of MODE, as described earlier. (TEMP) (MDBG) is used during file modification. When a word in a file must be modified, the modified file is first created as (TEMP) (MDBG), and then the original file is deleted and (TEMP) (MDBG) is renamed. The length of this file depends on the length of the file being modified. The file has permanent mode. (MDBG) BSS is created by MADBUG whenever loading is required. Its position in the new core image was discussed earlier. It contains the bootstrap for MADBUG and the patch area. It is one track long and has temporary mode. (MBGI) SAVED is a very short program which processes the input line blocks the user types while editing. It processes all the input line blocks associated with one edit request and reads in the following request before chaining back to MADBUG. It is usually one track long and is permanent mode.

A user core image may use the command buffers. A call to CHNCOM will not return control to MADBUG. MADBUG saves the command buffers and counter initially and restores them

when the user gives the QUIT request. MADBUG also treats the command buffers and counter as psuedo-machine conditions associated with each core image. The buffers are only lost on manual restart. A fresh core image has empty buffers.

By editing, the user modifies the MAD subprogram on which he is working. By inserting and removing breakpoints and by changing the values of variables, the user modifies the current user core image, (USER) SAVED. MADBUG does not change external files until the changes are logically needed. If the user uses EXECUTE to ask CTSS to process these files, he may want to insure that these logical modifications are made physically. To insure that the MAD subprogram being worked is modified physically, give a redundant WORK request using the name of the subprogram already being worked. Whenever a WORK request is given, the logical modifications associated with the subprogram previously being worked are made physically. To insure that the current user core image is modified physically, use a SAVE request. A user who cannot afford the added tracks can give an "execute delete" on the created SAVED file.

This variation between the physical and logical modifications provides some degree of safety to the user who carelessly makes gross incorrect modifications to one of his programs. If the user should accidently type a "d" as a request line for example, he should quit by hitting the break button twice in succession. This will prevent MADBUG from actually deleting the file in question.

SUMMARY OF MADBUG REQUESTS

request	arguments	additional lines (3)	page
work	subprogram name	none	8
print	card names (1)	card images by MADBUG	10
delete	card names (1)	none	11
insert	card names (1)	card images by user	11
change	card names (1)	card images by user	11
append	none	card images by user	12
	(or) subprogram names	none	
manipulate	special, then cards	card names by MADBUG	13
translate	none	comments by MADBUG	14
use	subprogram names	none	15
go	card name or none	comments by MADBUG (4)	16
open	variables (1,2)	values by both (4)	17
verify	variables (1,2,5)	values by MADBUG (4)	18
linkage	none	linkage by MADBUG (4)	19
break	card names (1)	none (4)	19
kill	card names (1)	none (4)	19
save	save-name	none (4)	19
restore	save-name	none (4)	20
quit	none	none	20
execute	command and arguments	depends on command	21
force	parameter, special	none	23

notes: (1) If none, all are implied. (p. 10, p. 15)
(2) Optional form forcing first argument. (p. 25)
(3) Any request can get error comments from MADBUG.
(4) Comments by MADBUG if core image is created. (p.16)
(5) There is an optional save-name argument. (p. 18)