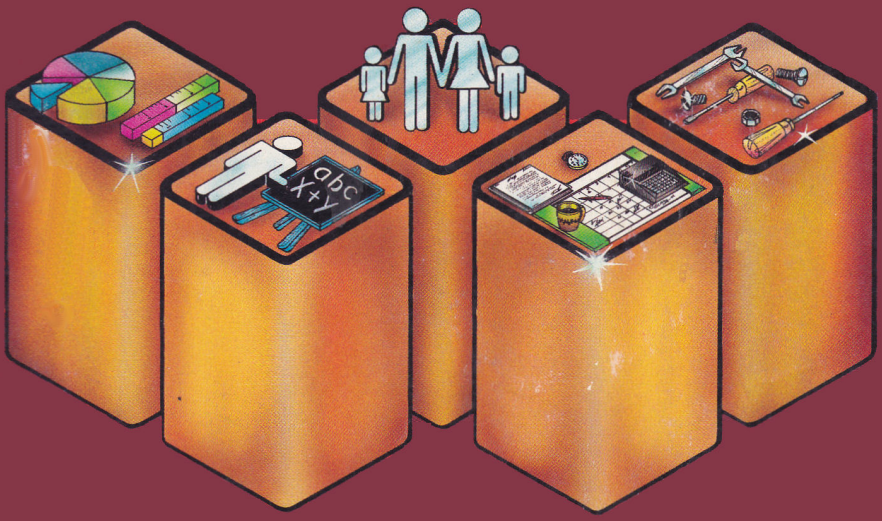
MUNCH A BUG™

A Machine Language Debugger
For The Apple II Family

By Wink Saville



Roger Wagner™
PUBLISHING, INC.

To re-enter MAB:
CALL %30467

MUNCH A BUG™

A Machine Language Debugger For The Apple II Family

By Wink Saville

INSTRUCTION MANUAL

ISBN 0-927796-07-4

PRODUCED BY:

Roger Wagner™
PUBLISHING, INC.

1050 Pioneer Way • Suite P • El Cajon, CA 92020
Customer Service And Technical Support 619/442-0522

OUR GUARANTEE

This product carries the unconditional guarantee of satisfaction or your money back. Any product may be returned to place of purchase for complete refund or replacement within thirty (30) days of purchase if accompanied by the sales receipt or other proof of purchase.

PRODUCT REFERENCE: MUNCH A BUG 2.5C0587

First, our legal stuff...

ROGER WAGNER PUBLISHING, INC.
CUSTOMER LICENSE AGREEMENT

IMPORTANT: The Roger Wagner Publishing, Inc. software product that you have just received from Roger Wagner Publishing, Inc., or one of its authorized dealers, is provided to you subject to the Terms and Conditions of this Software Customer License Agreement. Should you decide that you cannot accept these Terms and Conditions, then you must return your product with all documentation and this License marked "REFUSED" within the 30 day examination period following the receipt of the product.

1. License. Roger Wagner Publishing, Inc. hereby grants you upon your receipt of this product, a nonexclusive license to use the enclosed Roger Wagner Publishing, Inc. product subject to the terms and restrictions set forth in this License Agreement.

2. Copyright. This software product, and its documentation, is copyrighted by Roger Wagner Publishing, Inc. You may not copy or otherwise reproduce the product or any part of it except as expressly permitted in this License.

3. Restrictions on Use and Transfer. The original and any backup copies of this product are intended for your personal use in connection with a single computer. You may not distribute copies of, or any part of, this product without the express written permission of Roger Wagner Publishing, Inc.

LIMITATION ON WARRANTIES AND LIABILITY

ROGER WAGNER PUBLISHING, INC. AND THE PROGRAM AUTHOR SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO PURCHASER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY THIS SOFTWARE, INCLUDING, BUT NOT LIMITED TO ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THIS SOFTWARE. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

Then Apple's...

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

And now on with our program!

ABOUT THE AUTHOR

Wink Saville has a broad background in Computer Science and Electrical Engineering and has been responsible for many commercial software, hardware and firmware projects. In addition to writing Munch A Bug, Wink helped develop the AppliCard, and co-authored a book on Assembly Language routines. He built his own "KIM " computer from a kit in 1975 and three years later got his first Apple. A native Californian, Wink lives on the coast and enjoys both sailing and running on the beaches. But his greatest joy is found in time spent with his wife and children.

TABLE OF CONTENTS

INTRODUCTION

Munch-A-Bug (MAB) is a 6502 program which assists the user in debugging 6502 assembly language programs under the APPLE DOS operating system. MAB requires an Apple II computer, 1 APPLE disk drive and the APPLE DOS 3.3 operating system.

YOUR DISKETTE

Make a back-up of the MAB disk using any good copy program. Use only a back-up copy of MAB for debugging purposes as faulty assembly language programs can easily crash the system. To examine the files present on the diskette, place the back-up diskette in the drive and type in the word "CATALOG" and press RETURN. The catalog should appear as follows:

```
DISK VOLUME 254          260

 *A 005 HELLO

 MUNCH-A-BUG
 -----------
 *B 054 MAB
 *B 043 SMAB
 *B 054 MAB.D000

 SUPPORTING FILES
 ----------------
 *A 016 MAB CONFIGURE.A
 *T 002 DOUBLETIME ROM PATCH

 SAMPLE FILES
 ------------
 *B 002 TESTSYMBOLS
 *B 002 HANDYSYM.5000
 *B 002 MERLIN SYM EX.S

 LANG. CARD FILES
 ----------------
 *I 006 APPLESOFT
 *B 050 FPBASIC
```

MAB and MAB.D000 are the actual Munch-A-Bug program files. SMAB is a short version of MAB that will give a little more room for debugging programs with larger memory requirements. The CONFIGURE file is used to configure MAB to your particular situation in regards to memory usage.

TESTSYMBOLS is a sample file used in some of the examples within this manual. HANDYSYM.5000 is another sample file of a few labels which may be useful when using MAB's mini-assembler to create your own object code. The Language card files are included to allow Munch-A-Bug to boot properly on any configuration of Apple //e, Apple II+, or Apple II.

SPECIAL DOS FEATURES

Although you do not have to boot on the MAB diskette to use the program, you may wish to boot on the program disk so as to automatically install a special disk operating system (DOS) with some helpful features.

Specifically, on each CATALOG of a diskette, you will notice a number just to the right of the 'DISK VOLUME 254' message, similar to the '260' in the sample directory listed earlier. This is the number of free sectors remaining on the diskette being CATALOGed. A normal Apple DOS 3.3 diskette has a maximum of 496 free sectors available.

The second feature is an optional termination of the CATALOG listing at any of the pauses which usually occur when a directory has more files than can be displayed on the screen at one time. When using the modified DOS installed by the MAB diskette, pressing RETURN will terminate the CATALOG listing at any pause. Pressing any other key will continue it.

RUNNING Munch-A-Bug

MAB is installed and put in an active state in your computer by placing the MAB diskette in the current drive and typing in:

```
BRUN MAB
```

MAB will relocate itself between the DOS buffers, display the NEXT and END of available memory, and then prompt the user with an exclamation point "!". (Please note: throughout this document all numeric values are in hexadecimal).

```
MAB VER 2.6 (C) 1983 WINK SAVILLE
NEXT END
0800 7000
!
```

Now re-boot with PR#6

To use SMAB, a smaller version of MAB type in:

```
BRUN SMAB
```

This display should follow:

```
MAB VER 2.6 (C) 1983 WINK SAVILLE
NEXT END
0800 7900
```

You should notice the extra room. This extra space however is at the expense of a few features. Specifically, the assembler and symbol table commands are no longer available.

If you have a language card, and wish to install MAB on the card, you may type in:

```
BRUN MAB.D000
```

You should then see this display:

```
MAB VER 2.6 (C) 1983 WINK SAVILLE
NEXT END
0800 95FF
!
```

Note that using the language card version of MAB will erase any language currently residing on the RAM card. Specifically, if you have an Apple II Plus or an Apple //e, Integer BASIC will no longer be available. Likewise, if you have an Apple II with Integer BASIC on the motherboard, Applesoft will be deleted from the RAM card. This also applies to MERLIN if it is on the language card.

The language card version also uses significantly more zero page locations (28 contiguous bytes) and thus may present certain difficulties. The main reason MAB.D000 is provided is for debugging machine language programs that are independent of the usual Apple BASIC ROMs and which are too large to allow the normal version of MAB to be installed. See the section on configuring MAB for more details on the specific zero page byte usage of each version of MAB.

To make best use of this program, it is suggested that you first lightly read through the following description of the overall syntax and various commands available. Then go back and read the sections in which you're interested in greater detail later. Each command is provided with examples illustrating the details of its operation.

In presenting Munch-A-Bug, it is assumed that you have at least some familiarity with machine language programming, and presumably also a good assembler. It is beyond the purposes of this package to provide an in-depth tutorial on 6502 programming, or a complete assembler.

If you are in need of either an assembler or more information on machine language programming, Roger Wagner Publishing highly recommends the following products:

MERLIN - An extremely powerful 6502 macro assembler, yet simply designed for easy use by even novice programmers.

ASSEMBLY LINES: THE BOOK (by Roger Wagner)

This is an excellent starting point for anyone wanting to improve their machine language programming skills.

See the Roger Wagner Publishing Product Guide for more information, or call or write us at the address and phone number listed on the title page of this manual.

As with any tool, as you use Munch-A-Bug, you'll develop your own preferred techniques of use. We also appreciate any comments you may wish to make as to ways in which we can improve this product to better serve you.

M.A.B. COMMANDS

A) (1) Assemble 6502 mnemonics.
B) Block search.
C) Call a subroutine.
D) Dump memory.
F) Fill memory with a value.
G) (2) 6502 Go with optional breakpoints.
K) (1) Connect and delete symbol table.
L) List in 6502 assembly language mnemonics.
M) Move memory.
N) Execute Next instruction but do not trace JSR's.
O) Other screen display.
P) (2) Pass pointers: display, modify or delete.
Q) Quit MAB.
S) Substitute memory.
T) Trace a 6502 program.
U) Untrace a 6502 program (no display of steps).
V) Convert a number to decimal and hexadecimal.
W) Freeze the display Window to save data.
X) Examine, display, and modify the 6502 registers.
Y) (1) Symbol definition, display and modification.
;) Semi-colons are ignored and are treated as comments. (when 'EXEC'ing in files for example).

-> Note that any legal DOS command can also be directly executed at any time within MAB.

(1) Not available in SMAB.
(2) Not available when tracing code in ROM.

CONTROL KEYS

^Q Control Q is used to stop the current output operation. To resume normal operation, enter a second Control-Q. (Please note: If the ESC or RETURN key is pressed the output is stopped and the current operation terminated.

^Y Control Y is used to reenter MAB from the APPLE II monitor. The control Y entry point is at 3F8 and can also be used to reenter MAB from BASIC by CALLing 1016. (Note: language card version, ie. MAB.D000 cannot be re-entered after a Quit.)

^Z Control Z is used to permanently leave MAB (zap). This is used to remove MAB from between the DOS buffers. If MAB is left by a route other then ^Z, and can't be re-entered, then it is necessary to re-boot the system to clear out the space between the DOS buffers.

Items in {braces} are comments and are provided in examples for explanatory purposes only. They are not to be entered.

Items in <broken brackets> are optional.

Example:

D <"or´><ael><,ae2>

ae: is defined as an address expression with the following forms:

number
number + number
number - number

Where "number" is a hexadecimal, decimal or a symbol, optionally preceded by a plus or minus sign. An exclamation point (!) is also treated as a number, in that MAB will look at the last two bytes on the stack, and assume they form a return address for an RTS. See the ´G´ and ´T´ commands for more information on this. Hexadecimal is the default radix but may be preceded by a dollar sign "$". Decimal is always preceded by a percent sign (%). Symbols are preceded by a period. For example:

```
3000 = 3000
3000+10 = 3010
3000-10 = 2FF0
-3000+-100 = CF00
$3000+10 = 3010
%100 = 0064
-%10 = FFF6
.SYM1 = 5000
.SYM2 = 5010
.SYM1+.SYM2 = A010
.SYM2-10 = 5000
! = next address on the stack
```

It's easy to miss the notion that labels (including the exclamation point), once defined, can be used in place of numbers anywhere. The easiest way to learn what can be used is to just try something out whenever you think it might be possible. You'll probably be pleasantly surprised!

NOTE: For purposes of clarity throughout this manual, command characters are shown separated from their parameters by a single space. This space is optional and may be omitted when actually entering a command.

Next location:

This refers to the fact that some of the commands save the last location used. This will be the default if the first address expression, "<ael>", is omitted. Four commands have this feature: Dump, Assemble, Substitute, and Disassemble. For example:

```
!D 1100,110F
1100: 00 00 00 00-00 00 00 00 ........
1108: 00 00 00 00-00 00 00 00 ........

!D ,111F
1110: 00 00 00 00-00 00 00 00 ........
1118: 00 00 00 00-00 00 00 00 ........
```

Strings and characters:

There are two methods of defining characters. The first is a single quote ('). This defines the character with bit 7 = 0. The second method is a double quote ("), which defines the character with bit 7 = 1. Four commands allow the use of the quote: Dump, Assemble, Substitute and Fill.

Examples:

```
!S 1100
1100 00 "STRING WITH BIT 7 = 1
1115 00 'STRING WITH BIT 7 = 0
112A 00 .
```

```
!D ´1100,112F
1100: D3 D4 D2 C9-CE C7 A0 D7 ........
1108: C9 D4 C8 A0-C2 C9 D4 A0 ........
1110: B7 A0 BD A0-B1 53 54 42 .....STR
1118: 49 4E 47 20-57 49 54 48 ING WITH
1120: 20 42 49 54-20 37 20 3D  BIT 7 =
1128: 20 30 00 00-00 00 00 00  0.......

!D "1100,112F
1100: D3 D4 D2 C9-CE C7 A0 D7 STRING W
1108: C9 D4 C8 A0-C2 C9 D4 A0 ITH BIT
1110: B7 A0 BD A0-B1 53 54 42 7 = 1STR
1118: 49 4E 47 20-57 49 54 48 ING WITH
1120: 20 42 49 54-20 37 20 3D  BIT 7 =
1128: 20 30 00 00-00 00 00 00  0.......
```

>> TRACING COMMANDS <<

This section contains the instructions and examples needed to get started with MAB. The instructions for tracing code are presented first since that is the main reason for using MAB. It is suggested that the manual first be read in its entirety, but it is possible to start at the beginning of this section and proceed to work through the examples. If a previously unmentioned command is used to aid the example then either type what is in the manual or reference the command description elsewhere for clarificaton and then go back to the example.

TRACING COMMANDS

LIST (DISASSEMBLE TO 6502 MNEMONICS): L <ael><,ae2>

Disassemble memory starting at "ael" through "ae2". If "ael" is omitted then use the next location as the starting address. If "ae2" is omitted then disassemble one page. If the next byte to disassemble is not a 6502 opcode then a ??? will be printed. Note: At any time the listing may be aborted by typing any key, also control S and Q are operable.

This instruction is used to examine machine language programs in memory. The advantages of this command over the built-in "L" command in the Apple's Monitor are:

1. Constants are translated into their ASCII character equivalents where possible.

2. If a symbol table is connected (see the "K" and "Y" commands of MAB), then referenced addresses with labels will be displayed with their appropriate names.

3. You can disassemble more or less than the usual 20 instructions which would otherwise be required by the Monitor's LIST command.

4. You can list code by directly using the label name to specify the starting address, rather than having to remember the exact address itself. For example, L.ENTRY would be an acceptable use of MAB's "L" command.

Example:

First assemble some instructions at 1100.

```
!A 1100
1100 LDA #1
1102 LDA 10
1104 LDA 100
1107 LDA 10,X
1109 LDX 10,Y
110B LDA 100,X
110E LDA 100,Y
1111 LDA (10,X)
1113 LDA (10),Y
```

(Continued...)

```
1115 CLC
1116 ASL
1117 ASL 0A
1119 BCC 1100
111B JMP (200)
111E NOP
111F .
```

List 1100 through 1104.

```
!L 1100,1104
1100  A9 01     LDA #$01
1102  A5 10     LDA $10
1104  AD 00 01  LDA $0100
```

List from where we left off to 1113.

```
!L ,1113
1107  B5 10     LDA $10,X
1109  B6 10     LDX $10,Y
110B  BD 00 01  LDA $0100,X
110E  B9 00 01  LDA $0100,Y
1111  A1 10     LDA ($10,X)
1113  B1 10     LDA ($10),Y
```

List one page. NOTE: The data following the NOP at 111E may be anything including valid instructions. For our purposes, it will be assumed that the data is "garbage" and therefore the disassembler prints ???.

```
!L
1115  18        CLC
1116  0A        ASL
1117  06 0A     ASL $0A
1119  90 E5     BCC $1100
111B  6C 00 02  JMP ($0200)
111E  EA        NOP
111F  xx        ???
1120  xx        ???
1121  xx        ???
1122  xx        ???
```

TRACE A 6502 PROGRAM: <->T <ael><,ae2>

Trace "ael" number of instructions starting at location "ae2". If "ael" is omitted then trace one instruction and go into a keyboard wait loop; If "T" is pressed then trace the next instruction; If "U" then untrace; If "N" then trace the next instruction skipping JSRs; Pressing any other key will abort. If "ae2" is omitted then begin at the current program counter. The trace mode operates as follows, first the register and next instruction are displayed, then MAB waits for a keypress.

When a key is pressed, if the key is NOT a T, N or U, tracing is terminated. If the key is a T (for continue tracing. Key can also be a N or U - see later sections), the next instruction is executed, the registers displayed with the new data, and the next instruction after that displayed while MAB again waits for another keypress.

NOTE: The optional leading minus sign (-) turns off checking for an abort key from the keyboard so MAB does not hinder other keyboard checking from the user program during an extended (multi-step) trace.

Here are some examples of the Trace command:

Assemble a few instructions at 1100.

```
!A 1100
 1100 LDA C000
 1103 BPL 1100
 1105 BIT C010
 1108 JMP 1100
 110B.
```

Trace one instruction starting at location 1100. This enters the keyboard loop.

```
!T,1100
 ------ A=00 X=00 Y=00 S=FF P=1100
  LDA   $C000
   {Press a key other than T,U or N to exit loop.}
```

Trace two instructions starting at location 1100.

```
!T2,1100
 ------ A=00 X=00 Y=00 S=FF P=1100
  LDA   $C000
 ------ A=0D X=00 Y=00 S=FF P=1103
  BPL   $1100
```

Trace three instructions from the current 6502 program counter

```
!T3
 ------ A=0D X=00 Y=00 S=FF P=1100
  LDA   $C000
 ------ A=0D X=00 Y=00 S=FF P=1103
  BPL   $1100
 ------ A=0D X=00 Y=00 S=FF P=1100
  LDA   $C000
```

Trace the next instructions one at a time.

```
!T
 ------ A=0D X=00 Y=00 S=FF P=1103
  BPL   $1100       {PRESS 'T' TO TRACE NEXT STEP}
 ------ A=0D X=00 Y=00 S=FF P=1100
  LDA   $C000
{Press a key other than T,U or N to exit loop.}
```

Trace nine instructions but use a leading minus sign to allow the program we are testing to get the next character from the keyboard:

```
!-T9
 ------ A=0D X=00 Y=00 S=FF P=1100
  LDA   $C000
 ------ A=0D X=00 Y=00 S=FF P=1103
  BPL   $1100
 ------ A=0D X=00 Y=00 S=FF P=1100 ; HIT "A" HERE
  LDA   $C000
 N------ A=C1 X=00 Y=00 S=FF P=1103
  BPL   $1100
 N------ A=C1 X=00 Y=00 S=FF P=1105
  BIT   $C010
 N------ A=C1 X=00 Y=00 S=FF P=1108
  JMP   $1100
 N------ A=C1 X=00 Y=00 S=FF P=1100
  LDA   $C000
 ------ A=41 X=00 Y=00 S=FF P=1103
  BPL   $1100
 ------ A=41 X=00 Y=00 S=FF P=1100
  LDA   $C000
```

NOTE: Trace will NOT stop at the 'RTS' at the end of a routine if too many steps have been specified.

Example:

Assemble a short (2 step) routine at 1100

```
!A 1100
 1100 LDA #0
 1102 RTS
```

Trace THREE instructions

```
!T 3,1100

 ------ A=00 X=00 Y=00 S=FF P=1100
  LDA   #$00
 ----Z- A=00 X=00 Y=00 S=FF P=1102
  RTS
 ----Z- A=00 X=00 Y=00 S=01 P=??? {unpredictable}
  ???
```

One easy way to prevent "overruns" is to set a pass pointer equal to zero at the RTS address. For example:

```
!P 1102

!T 3,1100

 ------ A=00 X=00 Y=00 S=FF P=1100
  LDA   #$00
0000 PASS 1102
 ----Z- A=00 X=00 Y=00 S=FF P=1102
  RTS
```

If you are tracing code that calls subroutines, you may want to skip tracing the JSR's to subroutines that are known to work. This would normally be done with the 'N' command. If, however, you should accidentally enter a subroutine, you can return to the code you were executing immediately after the JSR by entering:

```
!G,!
```

The (second) exclamation mark tells MAB to execute code starting at the current address, without tracing, and to break at whatever address is currently on the stack as a return address. When the registers are displayed, resume tracing with the usual "T" keypress.

Trace, Go, and pass pointers can also be combined to directly trace code called from BASIC programs.

For example, with MAB up and running, enter this short program at 300:

```
!A 300
 300 LDA #$01
 302 RTS
```

Then set a pass pointer at 300:

```
!P 300          {-P turns pass pointers off}
```

Then exit MAB with a 'Q' (for quit) and enter the following Applesoft program:

```
10  HOME: INPUT"YOUR NAME?";I$
20  LIST : PRINT
30  CALL 768: GET A$: PRINT A$
40  IF A$ <> "X" THEN 30
```

And now RUN to see what happens. After the screen clears, you should be able to enter your name. As soon as you press RETURN, the program should list itself and do a CALL to the routine at 300. Since a pass pointer was set there, MAB will instantly pop up in the trace mode, with the MAB prompt. For now, enter just 'G' to verify that you can immediately return to the program. If you press any key other than 'X' in the BASIC program, each CALL 768 will trigger MAB's pass pointer.

When the pass pointer break occurs, you could also trace any portion of the code you desired, although in this case that would be rather limited. Also of interest is the fact that you can literally trace Applesoft by continuing to trace past the RTS.

Experiment with the different combinations of Go, Trace, Untrace, and pass pointers to see just what kinds of things are possible. MAB is excellent for debugging machine language routines called from BASIC. Remember to use 'G' to resume normal program operation from the Trace mode at any time.

ROUTINE MACHINE is an excellent example of the value in interfacing machine language routines to Applesoft. Basically, ROUTINE MACHINE is a pre-written library of machine language routines for programmers of all skill levels to use. It makes extensive use of routines called from BASIC. See the current Roger Wagner Publishing Product Guide for more information on this product.

The trace function is, for most people, the most used portion of MAB. It is through the use of this particular command that you will debug most programs. The idea behind debugging utilities such as MAB is to give the programmer the ability to watch each individual step of a machine language program. This is so that you can confirm that each step of your program does in fact do what you thought it was supposed to.

For example, consider this simple program to read the game controllers. The paddles on the Apple are read by loading the X Register with the value for the paddle you wish to read, followed by a JSR to $FB1E, a routine, cleverly enough, called PREAD (for Paddle READ). The result, a value between $00 and $FF, is returned in the Y Register. So that we can tell what the paddle is reading, we'll then transfer the contents of the Y register to the Accumulator, followed by a JSR COUT. You may be aware that COUT ($FDED) is the routine used by the Apple to print characters to the screen. When the program is run, we would expect different characters to be printed to the screen depending on the paddle position.

Suppose we had entered the following routine to read paddle #0:

```
300:A6 02       LDX $01     {set X = $01}
302:20 1E FB    JSR $FB1E   {read paddle}
305:98          TYA         {put in acc.}
306:20 ED FD    JSR $FDED   {print char.}
309:4C 00 03    JMP $300    {go back again}
```

If you enter this as an example, be SURE to enter it EXACTLY as shown!

Now, suppose when you run the program, all that appears on the screen are inverse "@" characters, regardless of the paddle position. (Rembember paddle #1 is actually your second paddle!) Something is wrong somewhere, but how to find it? The answer (of course!) is with MAB. With MAB installed, trace through the program to examine each step.

To do this, the first command will be:

```
!T,300
```

MAB should display:

```
----- A=00 X=00 Y=00 S=FF P=0300
 LDX '$01
```

This shows you the registers, and MAB now waits for a keypress to begin tracing through your program one step at a time. Press the 'T' key once to show the next step. MAB should now display:

```
----- A=00 X=4C Y=00 S=FF P=0300
 JSR  $FB1E
```

Upon examining the X Register, we notice that it has been set to '$4C', rather than '$01' as we had intended. Why? Aha! Looking at the first instruction, you can see that we omitted the immediate '#' character. Instead of loading X with the constant $01, we are actually loading the X Register with the CONTENTS of location $01! Now, using the 'A' (for assemble) command of MAB, re-write the first line to read:

```
300:A2 02        LDX #$01    {set X = $01}
```

(Remember that to use the assembler, you need only to enter:

```
!300: LDX #$01
!301: {RETURN alone to exit}
```

Now when the program is run, it should behave as expected.

MAB will be of its greatest value when you are SURE the program should be working a certain way and it isn't by allowing you to see what you REALLY wrote in the program!

UNTRACE A 6502 PROGRAM: <-> U <ael><,ae2>

Untrace is identical to Trace except that nothing is displayed during execution. This provides a means of still controlling the execution, but it executes much faster than Trace. Untrace may be aborted at any time by pressing any key. Upon termination of the untrace command, the registers will be displayed. This command is useful for locating the general address range and environmental conditions of an infinite loop.

The optional leading minus sign (-) turns off the feature of checking for an abort key from the keyboard so that MAB does not hinder other input code. See the section on MAB's trace command for use of the minus sign.

Example:

Assuming the same code as in Trace above, execute five instructions beginning at $1100.

```
!U 5,1100
 ------ A=0D X=00 Y=00 S=FF P=1100
  LDA   $C000
```

NOTE: Untrace will not stop at the 'RTS' at the end of a routine if too many steps have been specified. (See "Trace"). Again, use of a pass-pointer at the RTS will prevent an overrun of the RTS.

Example:

Assemble a short (2 step) routine at 1100

```
!A 1100
 1100 LDA #0
 1102 RTS
```

Untrace THREE instructions

```
!U 3,1100

 ------ {unpredictable data here}
  ???
```

NEXT INSTRUCTION: N <ael><,ae2>

This is the same as the Trace command except that subroutines are not traced. This is very useful when debugging routines which call subroutines and you know that the subroutine is operating correctly. An example might be when calling operating system routines.

Example:

Assemble a routine which will output a character to the screen via the monitor.

```
!A 1000
1000 LDA #"A"
1002 JSR FDED
1005 JMP 1000
1008 .
```

Use the N command to debug it:

```
!N 6,1000                              {START AT 1000}
 ------ A=00 X=00 Y=00 S=FF P=1000
  LDA   #$C1 "A
 N----- A=C1 X=00 Y=00 S=FF P=1002
  JSR   $FDED                  {CALL THE SUBROUTINE}
A -V--ZC A=C1 X=00 Y=00 S=FF P=1005
  JMP   $1000
 -V--ZC A=C1 X=00 Y=00 S=FF P=1000
  LDA   #$C1 "A
 NV---C A=C1 X=00 Y=00 S=FF P=1002
  JSR   $FDED
A -V--ZC A=C1 X=00 Y=00 S=FF P=1005
  JMP   $1000
```

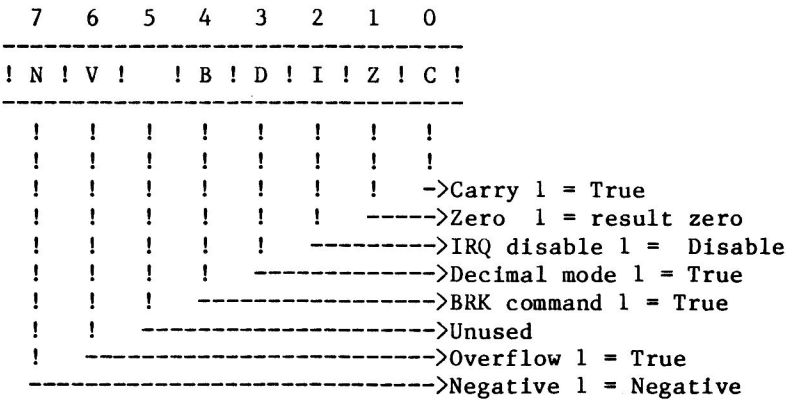
When tracing code that you expect will require the 'N' command to skip a JSR, use the N from the beginning, instead of 'T' (for Trace). They are identical, except for 'N's ability to skip JSR's. You can also use the 'W' (to freeze the Window) command to freeze a portion of the listing on the screen to see what the next instruction to be executed will be.

EXAMINE 6502 REGISTERS: X <reg>

Display or change the 6502 registers and the M.A.B. control registers. If "reg" is omitted then all of the 6502 registers are displayed. The following is the definition of the registers.

A	Accumulator
F	Flags

Bit Number:



X	X index register
Y	Y index register
S	Stack register
P	Program counter

M — This is a pseudo register which contains the last position of where M.A.B. may use the 6502 stack page at 100. The default value is 2F and may be changed but it should not be less than 2F.

C — This is a pseudo register which defines the size of the terminal screen. If C=40 then a 40 column screen is assumed, if C=80 then a 80 column screen is assumed. This will effect the Dump command, and the 6502 register display.

D This is a pseudo register which defines if the symbols in the symbol table with non-zero length bytes will be displayed when registers are dumped. This is either "T" true or "F" false. This is a very powerful option which allows the user a close watch on specific variables during tracing and pass pointer operations.

Examples:

Display all of the registers.

```
!X
 ------ A=FF X=00 Y=00 S=FF P=1100
  LDA #C1
```

Change register A to 1 and register P to 1200 then display the registers.

```
!XA
A=00<1

!XP
P=1100<1200

!X
 ------ A=01 X=00 Y=00 S=FF P=1200
  ???
```

The stack pointer, 'S' is unusual in that the 'XS' command is also used to examine the current stack contents. Suppose that this program has just been traced with the command T3,1100:

```
1100 LDA #01
1102 PHA
1103 PHA
1104 RTS
```

You can stop at 1104 and examine the stack by entering:

```
!XS
 01FD: xx 01 01
S=FD<??
```

Notice that the stack pointer always points to one byte PAST the last data byte on the stack. We can see the two '01' pushed onto the stack by the program. The contents of 1FD will be arbitrary. You can restore the stack to FF by entering FF and pressing RETURN:

```
S=FD<FF
!X
 ------ A=01 X=00 Y=00 S=FF P=1104
  ???
```

To protect its own stack area, MAB will prevent the stack pointer from staying in the range 0-2F (or whatever 'M' has been set to) by restoring the stack to FF whenever this occurs. This check is done each time the MAB prompt is presented.

Change the Column register to 80 and display the registers.

```
!XC
C=40<80

!X
 ------ A=01 X=00 Y=00 S=FF P=1200  ???
```

Change the Column register back to 40.

```
!XC
C=80<40
```

Change the Display register to TRUE.

```
!XD
D=F<T
```

Change the Display register back to FALSE.

```
!XD
D=T<F
```

Changing the flags is a little different. Each bit is controlled by its corresponding name <N,V,D,I,Z,C>. To turn a flag on simply type its name; to turn it off type a minus sign (-) followed by the name. To turn all of the flags off enter only a minus sign.

Turn on carry flag

```
!XF
------<C
```

Turn on the Z flag

```
!XF
-----C<Z
```

Turn on the N flag

```
!XF
----ZC<N
```

Turn off the C flag

```
!XF
N---ZC<-C
```

Turn off all the flags

```
!XF
N---ZC<-
```

QUIT M.A.B.: Q

This command sets the pass pointers (see the P command), prints an address to Call to re-enter MAB and then jumps to APPLE DOS via a jump to 3D0. The user may also reenter MAB by calling the control Y jump instruction at 3F8, which is 1016 decimal. This is how assembly language routines which are called by one of the BASICs may be debugged.

NOTE: Leaving MAB.D000 with this command is the same as leaving via the Control-Z command (Permanently). Do not attempt to re-enter MAB as the low memory locations are altered. You MUST BRUN MAB.D000 to restart it.

Example:

Return to APPLESOFT using the Q command

```
!Q
TO REENTER MAB -- CALL %30467

]
```

Return to MAB by calling 30467

```
]CALL 30467

!
```

ZAP M.A.B.: ^Z

This command is for the final exit from MAB to return the system to normal. Executing this command will reset the break vector and remove MAB from between the DOS buffers.

Example:

Return to APPLESOFT via the Control-Z command

```
!       {Type in CTRL-Z}

EXIT PERMANENTLY (Y/N)? Y

]
```

The following instructions are useful in calling a program when only part of it needs to be examined. This makes it possible to only trace certain sections of code, while the rest of the program runs at full speed. This, for example, allows an initalization routine to set up everything and drop into MAB when tracing is desired.

EXECUTION COMMANDS

CALL SUBROUTINE: C <ae>

This command calls a subroutine which begins at "ae". The subroutine should terminate with an RTS instruction to return control to MAB. Note that pass pointers are not checked during a call.

This may be used to add functions to MAB or for executing a particular user subroutine.

Example:

Place a subroutine at $1100 to increment location 0.

```
!A 1100
 1100 INC 0
 1102 BNE 1106
 1104 INC 1
 1106 RTS
 1109 .
```

Now at any time this subroutine could be called. Here we will zero locations 0 and 1, call the subroutine twice at $1100, and then display the contents of 0 which now contains a 2.

```
!S 0
 0000 FF 0     {ZERO LOCATION 0}
 0001 FF 0     {ZERO LOCATION 1}
 0002 .        {END SUBSTITUTION}
!C 1100        {CALL THE SUBROUTINE}
!C 1100        {CALL IT AGAIN}
!S 0           {DISPLAY CONTENTS OF LOCATION 0}
 0000 02 .
```

Call the Monitor HOME routine (FC58)

```
!C FC58
!              {screen clears and homes cursor}
```

If you have connected the HANDYSYM.5000 table (see 'K' cmd), use a label instead:

```
!C.HOME
!              {screen clears and homes cursor}
```

GO: G ael<,ae2><,ae3><,ae4>

Execute a 6502 program starting at "ael" with 3 optional (and temporary) breakpoints "ae2", "ae3", and "ae4". If "ael" is omitted then the current 6502 program counter is used as the starting address.

Multiple breakpoints are used when you may not be entirely sure which part of the program will eventually be executed, as might be the case when branch instructions are involved. THE BREAKPOINTS ARE ALL CLEARED IMMEDIATELY UPON RETURN OF THE PROMPT. For 'permanent' breakpoints, pass pointers with counters set to zero are used.

A 6502 BRK instruction is used for the breakpoints. M.A.B. regains control when a BRK instruction is executed or a pass pointer with a zero count is reached. Because ROM cannot be rewritten with breakpoints, this command may not function properly if used on routines residing in ROM assigned memory (usually $C000-FFFF). The Trace and Untrace functions will work properly even in ROM though, so you may wish to use these instead.

Example:

Place a "short" 6502 routine at 1100.

```
!A 1100
 1100 LDA #1
 1102 LDY #2
 1104 LDX #3
 1106 JMP 1100
 1109 .
```

Go at 1100 and set breakpoints at 1102 and 1106.

```
!G 1100,1102,1106
 ------ A=01 X=00 Y=00 S=FF P=1102
  LDY   #$02
```

Execute at current location and set a breakpoint at 1106.

```
!G ,1106
 ------ A=01 X=03 Y=02 S=FF P=1106
  JMP   $1100
```

The command phrase 'G,!' is also extremely useful for finishing out a subroutine you're in, and returning to the first instruction after the JSR that called the subroutine. See the Trace command for an example of this, and the advantages of combining the Go, Trace and Pass Pointer options.

PASS POINTERS: <->P <ae<,cnt>> or P ae,@subroutine

This command allows you to define, display, and modify pass pointers. A pass pointer is similar to a break point as used in the Go command except that pass pointers have a count field associated with them. They are also permanent to the extent that they remain in effect until you specifically clear them. When a pass pointer is defined you are telling MAB to set a break point at this location but only stop if the count field becomes zero. This is helpful when debugging subroutines which are called from BASIC or other assembly language code.

Note that pass pointers cannot be set to addresses within ROM assigned memory, such as in Applesoft or Integer. They CAN be used if the memory area corresponds to a RAM card.

The second form of the command, Pae,@subroutine, allows the user to define a subroutine which controls the stopping of the program at a pass point. Each time the program passes this type of pass pointer the subroutine is called. If the subroutine wants the debugger to stop execution it makes the Z flag 1; if it does not want MAB to stop, it makes the Z flag 0. Upon entry to the subroutine the registers A,P,X, and Y contain the values the program being debugged has. Also the APPLE register save area in page zero contains a copy of them. The APPLE register save area is:

3A,3B=Program Counter, 45=A, 46=X, 47=Y, 48=P, 49=S.

The following are valid P commands:

P	A "P" with no parameters displays all of the currently defined pass pointers followed by the current count
-P	This deletes all pass pointers
P ae	This defines a pass pointer at the address expression "ae" with a count field of 0. This is more or less equivalent to the conventional notion of a breakpoint, in that MAB will always stop here and display registers.

-P ae	This deletes the pass pointer at the address expression "ae".
P ae,cnt	This defines a pass pointer at the address expression "ae" with a count field of "cnt". MAB will display all registers on each pass, but will not stop until the counter reaches '0'.
P ae,@routine	This defines a pass pointer at the address expression "ae" with a subroutine at an address given by "routine".

Example:

Define a main program at 1100 which loads register A with 0 and then calls a subroutine at 1200 three times and then stops. We will set a pass pointer at 1200 with a count of 5 and execute the main program.

Define the main program and the subroutine:

```
!A 1100
 1100 LDA #0
 1102 JSR 1200
 1105 CMP #03
 1107 BNE 1102
 1109 RTS
 110A .

!A 1200
 1200 TAX
 1201 INX
 1202 TXA
 1203 RTS
 1204 .
```

Define the pass pointer at 1200, and another at 1109 to prevent an "overrun" of the last RTS at 1109.

```
!P 1200,5

!P 1109
```

Display the pass pointers

```
!P
 0005 1200
 0000 1109
```

Execute the main program and watch the results

```
!G 1100
0005 PASS 1200
 ----Z- A=00 X=03 Y=00 S=FD P=1200
  TAX
0004 PASS 1200
 N----- A=01 X=01 Y=00 S=FD P=1200
  TAX
0003 PASS 1200
 N----- A=02 X=02 Y=00 S=FD P=1200
  TAX
0000 PASS 1109
 ----ZC A=03 Z=03 Y=00 S=FF P=1109
  RTS
```

Display the pass pointers and notice that 1200 is now 2

```
!P
 0002 1200
 0000 1109
```

Now call it again, and notice that this time it stops at 1200, when the first pass pointer reaches '0'.

```
!G 1100
0002 PASS 1200
 ----ZC A=00 X=03 Y=00 S=FD P=1200
  TAX
0001 PASS 1200
 N----- A=01 X=01 Y=00 S=FD P=1200
  TAX
0000 PASS 1200
 N----- A=02 X=02 Y=00 S=FD P=1200
  TAX
```

We will now change the pass pointer at 1200 from being a counting type pass pointer to a subroutine type pass pointer. We'll also modify the routine so that it loops 255 times. It will now stop when register A becomes 81.

```
!A 1300                 {define the subroutine}
                        {enters with value in acc.}
 1300 CMP #81           {will return w/ Z clr until
 1302 RTS                acc. = #$81}
 1303 .

!P1200,@1300            {define the pass pointer}

!P                      {display the pass pointers}
@1300 1200
 0000 1109

!A1105                  {modify routine}
 1105 CMP #FF           {will loop FF times}
 1107 .

!G1100                  {execute}
@1300 PASS 1200
 N----- A=81 X=81 Y=00 S=FB P=1200
  TAX
```

In this case, it was the fact that the accumulator was equal to a specific value that triggered the pass pointer. You can also test in a similar manner for the contents of any other register or memory location:

```
!A 1300
 1300 CPX #02           {test for X = #02}
 1302 RTS
 1303 .

!A 1300
 1300 CPY #02           {test for Y = #02}
 1302 RTS
 1303 .
```

```
!A 1300
 1300 LDA 1000 {tests for location 1000 = #02}
 1303 CMP #02
 1305 RTS      {acc automatically restored
 1306 .        by MAB before returning...}
```

You can also test for a combination of conditions

```
!A 1300
 1300 CPX #02  {tests for X=2 AND Y=FF!}
 1302 BNE 1306
 1304 CPY #FF
 1306 RTS
 1307 .
```

Or...

```
!A 1300
 1300 CPX #02  {tests for X=$02 OR Y=$FF}
 1302 BNE 1305
 1304 RTS
 1305 CPY #FF
 1307 RTS
 1308 .
```

We will now delete all of the pass pointers.

```
!-P

!P      {See what's left...}

{no pointers left}
```

>> MEMORY MANIPULATION COMMANDS <<

These commands allow the user to examine and make changes to memory. These are useful commands to see how things are before and after a machine language program is run. They are also useful for just snooping around memory. These instructions include the ability to assemble 6502 mnemonics, dump, substitute, move, fill, and block search memory.

MEMORY MANIPULATION COMMANDS

ASSEMBLE 6502 MNEMONICS: A <ae>

Assemble in 6502 mnemonics beginning at "ae". If "ae" is omitted, then the next location is used. You will be prompted with an address and then MAB will wait for a valid mnemonic and operand. To leave the assembly mode, type a period (.), or press RETURN alone.

By typing a "^" character (shift-N = ^ on the Apple II, shift-6 = ^ on the Apple //e) you may back up to, and thus correct, the previous instruction. (Note that for a given entry, shift-N (^) only works the first time it's used. That is to say that subsequent entries of a shift-N will not decrement the address counter any further.)

Examples:

```
!A 1100
 1100 LDA #1          ;immediate addressing
 1102 LDA 10          ;page zero absolute
 1104 LDA 100         ;absolute
 1107 LDA 10,X        ;page zero indexed by X
 1109 LDX 10,Y        ;page zero indexed by Y
 110C LDA 100,X       ;indexed by X
 110F LDA 100,Y       ;indexed by Y
 1112 LDA (10,X)      ;indirect by X
 1114 LDA (10),Y      ;indirect by Y
 1116 CLC             ;implied
 1117 ASL             ;register A
 1118 ASL A           ;location $0A
 111A BCC 1100        ;relative
 111C JMP (200)       ;indirect
 111F NOP
 1120 ^               ;backup one instruction
 111F LDA #"A         ;character with bit 7 = 1
 1121 LDA #'A         ;character with bit 7 = 0
 1123 .               ;exit
```

Also acceptable:

```
     1102 LDA .LABEL
or
     1114 LDA (.LABEL),Y
```

When 'LABEL' has be defined using the 'K','KM', or 'Y' commands.

DUMP MEMORY: D <"or'><ael><,ae2>

Dump memory as Hex bytes and ASCII characters beginning at "ael" through "ae2". If a double quote follows the D, then bit 7 is ignored when displaying the ASCII. If a single quote or no quote follows the D, all bits are used to determine a valid ASCII character. If "ael" is omitted then the next location is used as the starting address. If "ae2" is omitted then one page of memory is dumped. Note: At any time the dump may be aborted by typing any key, also control S and Q are operable.

Example:

Place 2 strings in memory, the first with bit 7 = 1 the second with bit 7 = 0.

```
!S 1100
 1100 00 "STRING WITH BIT 7 = 1
 1115 00 'STRING WITH BIT 7 = 0
 112A 00 .
```

Dump from 1100 through 112F with bit 7 ignored for ASCII dump.

```
!D 1100,112F
 1100: D3 D4 D2 C9-CE C7 A0 D7 STRING W
 1108: C9 D4 C8 A0-C2 C9 D4 A0 ITH BIT
 1110: B7 A0 BD A0-B1 53 54 42 7 = 1STR
 1118: 49 4E 47 20-57 49 54 48 ING WITH
 1120: 20 42 49 54-20 37 20 3D  BIT 7 =
 1128: 20 30 00 00-00 00 00 00  0.......
```

Dump from 1100 through 112F with bit 7 ignored for ASCII dump.

```
!D "1100,112F
 1100: D3 D4 D2 C9-CE C7 A0 D7 STRING W
 1108: C9 D4 C8 A0-C2 C9 D4 A0 ITH BIT
 1110: B7 A0 BD A0-B1 53 54 42 7 = 1STR
 1118: 49 4E 47 20-57 49 54 48 ING WITH
 1120: 20 42 49 54-20 37 20 3D  BIT 7 =
 1128: 20 30 00 00-00 00 00 00  0.......
```

Dump from 1100 through 112F with bit 7 used for ASCII dump.

```
!D ´1100,112F
 1100: D3 D4 D2 C9-CE C7 A0 D7 ........
 1108: C9 D4 C8 A0-C2 C9 D4 A0 ........
 1110: B7 A0 BD A0-B1 53 54 42 .....STR
 1118: 49 4E 47 20-57 49 54 48 ING WITH
 1120: 20 42 49 54-20 37 20 3D  BIT 7 =
 1128: 20 30 00 00-00 00 00 00  0.......
```

Dump from where the previous dump stopped through 114F.

```
!D ,114F
 1130: 00 00 00 00-00 00 00 00 ........
 1138: 00 00 00 00-00 00 00 00 ........
 1140: 00 00 00 00-00 00 00 00 ........
 1148: 00 00 00 00-00 00 00 00 ........
```

SUBSTITUTE MEMORY: S <ae>

Change memory starting at location "ae". If "ae" is omitted then use the next location. The user will be prompted with the next address and its current value. Five operations are allowed:

1. If the value is a PERIOD, EXIT substitute mode.

2. If the value is a "^" character, BACK UP one byte.

3. If the value is a NUMBER, CHANGE the value and advance next location.

4. If the value begins with a DOUBLE QUOTE ("), enter a string until a carriage return is encountered and BIT 7 of each character will be SET (BIT 7 = 1).

5. If the value begins with a SINGLE QUOTE (´), enter a string a string until a carriage return is encountered and BIT 7 of each character will be CLEAR (BIT 7 = 0).

6. If NO ENTRY then ADVANCE to the next location.

Examples:

```
!S 3000                  {substitute starting at 3000}
 3000 00                 {do not change advance to text}
 3001 10 B               {change to 0B}
 3002 20 ^               {backup}
 3001 0B "A STRING 7=1   {change to string, bit 7 = 1}
 300D 23 ´A STRING 7=0   {change to string, bit 7 = 0}
 3019 30 23              {change to 23}
 301A 45 24^             {change to 24 and back up}
 3019 23 21              {change to 21}
 301A 24 .               {exit}
```

MOVE MEMORY: M ael,ae2,ae3

Move memory starting at "ael" through "ae2" to "ae3". Move may be aborted by pressing any key.

Example:

First fill 1100 through 111F with ´A´.

```
!F 1100,111F,´A
```

Now move 1100 through 111F to 3000.

```
!M 1100,111F,3000
```

Dump 3000 through 301F to verify move.

```
!D 3000,301F
3000: 41 41 41 41-41 41 41 41 AAAAAAAA
3008: 41 41 41 41-41 41 41 41 AAAAAAAA
3010: 41 41 41 41-41 41 41 41 AAAAAAAA
3018: 41 41 41 41-41 41 41 41 AAAAAAAA
```

NOTE: Unlike the Apple Monitor move routine, this routine will properly move a block of memory "up" in memory when the destination block overlaps the source block.

FILL MEMORY WITH A VALUE: F ael,ae2,v

Fill memory starting at "ael" through "ae2" with **the byte** value v. The value, v, must be a byte and may **be either** a numeric or a character. Fill may be aborted **by pressing** any key.

Example:

Fill 1100 through 110F with the value of A.

```
!F 1100,110F,A
```

Fill 1110 through 111F with the character 'A' (bit 7 = 0)

```
!F 1110,111F,'A
```

Fill 1120 through 112F with the character "A" (bit 7 = 1)

```
!F 1120,112F,"A
```

Dump to show result:

```
!D "1100,112F       {bit 7 ignored}

1100: 0A 0A 0A 0A-0A 0A 0A 0A ........
1108: 0A 0A 0A 0A-0A 0A 0A 0A ........
1110: 41 41 41 41-41 41 41 41 AAAAAAAA
1118: 41 41 41 41-41 41 41 41 AAAAAAAA
1120: C1 C1 C1 C1-C1 C1 C1 C1 AAAAAAAA
1128: C1 C1 C1 C1-C1 C1 C1 C1 AAAAAAAA
```

Dump to show result of using a single quote:

```
!D '1100,112F       {only bit 7=0 shown as ASCII}

1100: 0A 0A 0A 0A-0A 0A 0A 0A ........
1108: 0A 0A 0A 0A-0A 0A 0A 0A ........
1110: 41 41 41 41-41 41 41 41 AAAAAAAA
1118: 41 41 41 41-41 41 41 41 AAAAAAAA
1120: C1 C1 C1 C1-C1 C1 C1 C1 ........
1128: C1 C1 C1 C1-C1 C1 C1 C1 ........
```

BLOCK SEARCH: B ael,ae2,ae3,ae4

This command is used to search for a pattern of data in memory. The address pair ael,ae2 defines the range in memory to search, while the pair ae3,ae4 defines the beginning and ending addresses of the data buffer that holds the search data.

As an example let's first search for a string. First, use the substitute command to place the string in the memory block to be searched.

```
!S 1100
 1100 13 "TEST
 1104 00 .

!S 1200
 1200 85 "TEST
 1204 14 .
```

Now place "TEST" in the search buffer:

```
!S 300 D4 "TEST
   304 00 .
```

Perform the block search:

```
!B 1100,1210,300,303
1100
1200
```

Now for a second example. Search for three nulls:

```
!F 1100,1102,0              {Fill with three nulls}

!F 1180,1182,0

!F 300,302,0
```

Now perform the block search:

```
!B1100,1200,300,302
1100
1180
```

This last group of commands includes all the handy things that don't fit into the previous catagories. Imagine using a symbol table in the tracing of a program. The HEX-DEC converter is nice when dealing with BASIC. Also freezing the display window to keep track of part of a program is invaluable at times.

ADDITIONAL FEATURES

CONVERT A NUMBER TO HEXADECIMAL AND DECIMAL: V ae

Convert the number "ae" to hexadecimal and decimal. This is done by entering the number following the 'V' command. Numbers are assumed to be in hexadecimal form unless preceded by a '%' sign, which indicates a decimal number.

In addition, numeric expressions consisting of various numbers and labels are allowed.

Example:

```
!V100
$0100 %256


!V-100
$FF00 -%256


!V%100
$0064 %100


!V-%256
$FF00 -%256


!V-%253+-%10
$FEF9 -%263


!V.HOME-8
$FC50 -%984
```

FREEZE APPLE DISPLAY WINDOW: W <ae>

Pressing the 'W' key will freeze the top 12 lines of the standard Apple 40 column screen. This is useful for preserving various kinds of data on the screen, while you continue to trace a subroutine.

'W' acts as a "toggle", in that the first time you use it, it will freeze the top of the screen. The second time you press 'W', the screen scrolling will be restored to normal. The window is protected by modifying the contents of location $22, (WNDTOP = 34 decimal).

For example, to save part of a listing for future reference while later tracing it:

First, list the range with the 'L' command

```
!L 1100
1100  A9 00      LDA #$00
1102  A0 00      LDY #$00
1104  A2 00      LDX #$00
1106  20 0C 10   JSR $100C
1109  D0 FB      BNE $1106
110B  00         BRK
110C  18         CLC
110D  69 01      ADC #$01
110F  C8         INY
1110  E8         INX
```

Now, you can either: 1) press RETURN five times to advance the text to the top of the screen, and then press 'W', or, 2) Enter C.HOME if you've got HANDYSYM installed (or CFC58 if you don't) BEFORE listing the range, (to clear the screen and home the cursor), and then press 'W'. After pressing 'W', future text output will still continue to scroll out of the current window.

The other use for 'W' is to freeze one or more program step display lines. This is useful if you want to remember the 6502 registers at one point in a program, for later reference as you continue to trace the listing. With the 'W' command, you can freeze the top of the screen with the register display on it, and then compare all future steps against the frozen display.

You can also specify a value in the range 0-16 (0-22 decimal), which will freeze exactly that number of lines on the screen. MAB will also automatically adjust the number of lines output for the Dump, List, etc. commands accordingly. This can be useful even on 80 column cards where you might wish to limit the number of lines output at a time per command.

OTHER SCREEN DISPLAY CONTROL: O <L><H><G><T><M><F><1><2>

This command allows control of the Apple II screen. Any number of the command characters may be used and in any order. The following shows what each of the characters do:

L Sets the graphics mode to Low Resolution graphics.

H Sets the graphics mode to High Resolution graphics.

G Sets the screen mode to Graphics display.

T Sets the screen mode to Text display.

M Sets the graphics mode to Mixed (text/gr) screen display.

F Sets the graphics mode to Full screen display.

1 Sets the page display to Page 1.

2 Sets the page display to Page 2.

Example:

```
!OLGM1
```

Sets the screen to low resolution graphics, mixed text and graphics, page 1.

```
!OH  Switch now to HIRES (page 1, mixed by previous
     condition.)

!OT  (Go back to page 1 text mode.)

!O2  (Take a peek at page 2.)

!O1  (Go back again to page 1.)
```

CONNECT AND DELETE SYMBOL TABLES: K ae or -K

This command has three forms as follows:

K ae	Connect a list of symbols at "ae" onto the end of the table.
KM	Connect a Merlin symbol table. The Merlin symbol table must be present (see MAB and Merlin)
-K	Delete the entire symbol table.

The symbol table is composed of any number of symbol lists. Each list is composed of any number of symbol entries. Each symbol entry consists of two bytes of symbol value, in the normal 6502 order low byte, high byte; one byte defining the length of the symbol, and followed by the name, which is made up of ASCII characters terminated with a byte of zero. A list is terminated by a symbol value of zero and name of zero length. (ie. four '0's). In addition, the first character of a symbol name must be an alpha character and the last character must be an alpha or numeric character.

NOTE: If you are ever inclined to modify a table directly in memory, such as with the Substitute command, the table MUST be disconnected with a '-K' command first. Reconnect the table when you have completed the modifications.

There are three general types of symbols. These are LABEL, DATA, and INDIRECT symbols.

A LABEL is a symbol which identifies a pure memory location, such as the beginning of a routine or a hardware byte, such as the keyboard location ($C000). It's length byte in the symbol table is '0'.

A DATA symbol has a length of 1-127 associated with it, and denotes a part of memory that contains actual data. An example might be a part of memory that contains either a vector, pure data, or even a string. The length byte in the symbol table must have the high bit clear (bit 7 = 0) to denote this type of symbol.

An INDIRECT symbol is used to designate a pair of zero page bytes that point to another location in memory, such as would be used by an indirect addressing operation (i.e. LDA (PTR),Y). An "at" (@) sign is used when specifying this type of label instead of the shift-N (^). The length of the data field pointed at by the indirect symbol can be in the range of 1 to 127. An indirect symbol is defined by having the high order bit (bit 7) of its length byte set.

For more information regarding the use of the symbol table please see the "Y" command. The following listing illustrates the data structure of the symbol table itself, and the use of the 'K' command.

Examples:

Assume that the following code resides on the default drive as TESTSYMBOLS. (Such a file is present on the MAB diskette.)

```
1100:A91E    BEGIN    LDA  #$1E
1102:8506             STA  $06
1104:A911             LDA  #$11
1106:8507             STA  $07        ;PTR($06,07)=111E
1108:A000             LDY  #$00
             ;
110A:B106    LOOP     LDA  ($06),Y    ;(PTR),Y
110C:8D1D11           STA  MEMORY
110F:20EDFD           JSR  $FDED
1112:AD1D11           LDA  MEMORY
1115:C98D             CMP  #$8D       ;<CR>
1117:F003             BEQ  DONE
1119:C8               INY
111A:D0EE             BNE  LOOP
111C:60      DONE     RTS
             ;
111D:00      MEMORY   HEX  00         ;MEMORY LOCATION
111E:D4C5D3  STRING   ASC  "TEST"     ;TEST STRING
1122:8D               HEX  8D         ;TERMINATOR
             ;
```

```
1123:0011    TABLE    DA    BEGIN      ;DEFINE SYMBOL
1125:00               HEX   00         ;LENGTH = '0'
1126:C2C5C7           ASC   "BEGIN"    ;CHARACTERS OF NAME
112B:00               HEX   00         ;TERMINATOR = '0'
             ;
112C:0A11             DA    LOOP       ;DEFINE SYMBOL
112E:00               HEX   00         ;LEN = 0
                                       ;LABEL SYMBOL
112F:CCCFCF           ASC   "LOOP"     ;NAME
1133:00               HEX   00         ;TERMINATOR
             ;
1134:0600             DA    PTR        ;DEFINE SYMBOL
1136:85               HEX   85         ;LEN = 5 + HIGH BIT
                                       ;INDIRECT SYMBOL
1137:D0D4D2           ASC   "PTR"      ;NAME
113A:00               HEX   00         ;TERMINATOR
             ;
113B:1D11             DA    MEMORY     ;DEFINE SYMBOL
113D:01               HEX   01         ;LENGTH = 1
                                       ;DATA SYMBOL
113E:CDC5CD           ASC   "MEMORY"   ;NAME
1144:00               HEX   00         ;TERMINATOR
             ;
1145:0000             HEX   0000       ;SYMBOL ADDR = 0
1147:00               HEX   00         ;LENGTH = 0
1148:00               HEX   00         ;NO NAME (0)
```

Read in TESTSYMBOLS

```
!BLOAD TESTSYMBOLS
```

Call the routine to see what it does:

```
!C1100
 TEST              {routine prints the word "TEST"}

!
```

First let's see how the code would ordinarily disassemble:

```
!L1100,1122
 1100  A9 1E      LDA #$1E
 1102  85 06      STA $06
 1104  A9 11      LDA #$11
 1106  85 07      STA $07
 1108  A0 00      LDY #$00
 110A  B1 06      LDA ($06),Y
 110C  8D 1D 11   STA $111D
 110F  20 ED FD   JSR $FDED
 1112  AD 1D 11   LDA $111D
 1115  C9 8D      CMP #$8D
 1117  F0 03      BEQ $111C
 1119  C8         INY
 111A  D0 EE      BNE $110A
 111C  60         RTS
 111D  8D D4 C5   STA $C5D4
 1120  D3         ???
 1121  D4         ???
 1122  8D 00 11   STA $1100
```

And check the symbol table to see if there's anything there:

```
!Y
!{no table present}
```

Connect the symbol table in TESTSYMBOLS to the debugger

```
!K 1123 {THE SYMBOL TABLE BEGINS AT 1123}
```

Display all of the symbols in the symbol table

```
!Y
1100     BEGIN
110A     LOOP
0006@05 PTR
111D^01 MEMORY
```

Examine the contents of the data symbols:

```
!Y^
PTR@
 111E: D4 C5 TE
 1120: D3 D4 8D ST.
MEMORY
 111D: 8D .
```

These symbols will be displayed during Assemblies, Substitutions, Dumps, Dissassembly, and Pass Pointers. Following is an example of a dissassembly (with symbols) of TESTSYMBOLS.

```
!L1100,1122
BEGIN
 1100  A9 1E      LDA #$1E
 1102  85 06      STA $06       PTR
 1104  A9 11      LDA #$11
 1106  85 07      STA $07
 1108  A0 00      LDY #$00
LOOP
 110A  B1 06      LDA ($06),Y   PTR
 110C  8D 1D 11   STA $111D     MEMORY
 110F  20 ED FD   JSR $FDED
 1112  AD 1D 11   LDA $111D     MEMORY
 1115  C9 8D      CMP #$8D
 1117  F0 03      BEQ $111C
 1119  C8         INY
 111A  D0 EE      BNE $110A     LOOP
 111C  60         RTS
MEMORY
 111D  8D D4 C5   STA $C5D4
 1120  D3         ???
 1121  D4         ???
 1122  8D 00 11   STA $1100     BEGIN   {false
                                        assignment}
```

We can also delete the symbol table and then display the results.

```
!-K

!Y

{No Symbols...}
```

In general then, any table which has been constructed in, or loaded into memory can be connected to MAB's symbol table. This can be very useful when debugging programs of a large size where one cannot keep track of all the various subroutines and entry points.

If your assembler supports conditional assembly, one technique is to create the symbol table in the initial writing and debugging stages. Once the code is operational, the flag can be set "false" to suppress table generation in the final assembly.

DEFINE, DISPLAY AND MODIFY SYMBOLS:

```
Y<^ or @ length>
   or
Yname
   or
Yname,ae1<^ or @ length><,ae2>
```

This command allows definition, display and modification of symbols in the symbol table. Notice that symbols can have an associated length. The length parameters are specified with a (^) shift-N on the II or a shift-6 on the //e. This defines what is called a DATA SYMBOL, that is a name associated with a block of data somewhere. Symbols modified with an at sign (@) length parameter are called INDIRECT LABELS and have the advantage that data displayed will be in terms of where the two byte pair POINTS TO, not what they themselves contain. The length specified tells MAB how long the data field pointed to is.

Y	Display the entire symbol table.
Y^ (or just '^')	Display the locations pointed to by the symbols which have length bytes not equal to zero.
Y^length	Change the length field in every symbol to "length". Length must be a byte.
Yname	Display and allow modification of the symbol with the name of "name". The debugger will display the current value, length and allow modification of either or both the value and the length.

Yname,ael<^or@length><,ae2> Define a new symbol with the name of "name" the value of "ael" and the length of "length". The "ae2" value is a location that defines where to store the symbol data. ae2 is required if this is the first symbol you have defined. If a table already exists, ae2 is optional, and is to be used only when you want the symbol to be placed somewhere distinctly different than the end of the current table.

Examples:

Assume that the following code resides on the default drive as TESTSYMBOLS. (Such a file is present on the MAB diskette.)

```
1100:A91E       BEGIN   LDA  #$1E
1102:8506               STA  $06
1104:A911               LDA  #$11
1106:8507               STA  $07       ;PTR($06,07) = 111E
1108:A000               LDY  #$00
                ;
110A:B106       LOOP    LDA  ($06),Y   ;(PTR),Y
110C:8D1D11             STA  MEMORY
110F:20EDFD             JSR  $FDED
1112:AD1D11             LDA  MEMORY
1115:C98D               CMP  #$8D      ;<CR>
1117:F003               BEQ  DONE
1119:C8                 INY
111A:D0EE               BNE  LOOP
111C:60         DONE    RTS
                ;
111D:00         MEMORY  HEX  00        ;MEMORY LOCATION
111E:D4C5D3     STRING  ASC  "TEST"    ;TEST STRING
1122:8D                 HEX  8D        ;TERMINATOR
                ;
1123:0011       TABLE   DA   BEGIN     ;DEFINE SYMBOL
1125:00                 HEX  00        ;LENGTH = '0'
1126:C2C5C7             ASC  "BEGIN"   ;CHARACTERS OF NAME
112B:00                 HEX  00        ;TERMINATOR = '0'
```

```
112C:0A11               DA    LOOP      ;DEFINE SYMBOL
112E:00                 HEX   00        ;LEN = 0
                                        ;LABEL SYMBOL
112F:CCCFCF             ASC   "LOOP"    ;NAME
1133:00                 HEX   00        ;TERMINATOR
            ;
1134:0600               DA    PTR       ;DEFINE SYMBOL
1136:85                 HEX   85        ;LEN = 5 + HIGH BIT
                                        ;INDIRECT SYMBOL
1137:D0D4D2             ASC   "PTR"     ;NAME
113A:00                 HEX   00        ;TERMINATOR
            ;
113B:1D11               DA    MEMORY    ;DEFINE SYMBOL
113D:01                 HEX   01        ;LENGTH = 1
                                        ;DATA SYMBOL
113E:CDC5CD             ASC   "MEMORY"  ;NAME
1144:00                 HEX   00        ;TERMINATOR
            ;
1145:0000               HEX   0000      ;SYMBOL ADDR = 0
1147:00                 HEX   00        ;LENGTH = 0
1148:00                 HEX   00        ;NO NAME (0)
```

Read in TESTSYMBOLS

```
!BLOAD TESTSYMBOLS
```

And run the program to set up the INDIRECT SYMBOLS:

```
!C 1100
 TEST
```

Then connect the symbol table in TESTSYMBOLS to MAB

```
!K 1123          {the symbol table starts at 1123}
```

Display all of the symbols in the symbol table

```
!Y
1100     BEGIN
110A     LOOP
0006@05 PTR
111D^01 MEMORY
```

Add COUT (FDED) as a new symbol to the symbol table. This symbol refers to a routine in the Monitor and is referenced at 110F of the TESTSYMBOLS routine.

```
!Y COUT,FDED
```

Display all of the symbols in the symbol table

```
!Y
 1100     BEGIN
 110A     LOOP
 0006@05  PTR
 111D^01  MEMORY
 FDED     COUT
```

Add DONE as a new symbol to the symbol table. This symbol will be located at the RTS instruction of the LOOP subroutine.

```
!Y DONE,111C
```

Modify DONE to be at the BEQ instruction in the LOOP subroutine and then display the symbol table.

```
!Y DONE
111C   DONE<1117

!Y
 1100     BEGIN
 110A     LOOP
 0006@05  PTR
 111D^01  MEMORY
 FDED     COUT
 1117     DONE
```

We can also define a new list somewhere else in memory. Here we will define ASYM with a value of 1180 and a length of 5 at 2000 and then display the entire symbol table.

```
!Y ASYM,1180^5,2000

!Y
 1100     BEGIN
 110A     LOOP
 0006@05  PTR
 111D^01  MEMORY
 FDED     COUT
 1117     DONE
 1180^05  ASYM
```

Display the memory area of the symbols which have non-zero lengths.

```
!Y^
PTR@
 111E: D4 C5 TE.
 1120: D3 D4 8D ST.
MEMORY
 111D: 8D .
ASYM
 1180: 01 02 03 04-05 .....
```

Change all of the symbol table lengths to 1

```
!Y^1

!Y                {display the results}
 1100^01 BEGIN
 110A^01 LOOP
 0006^01 PTR
 111D^01 MEMORY
 FDED^01 COUT
 1117^01 DONE
 1180^01 ASYM
```

Change the value and length of ASYM to 1300 and 2

```
!Y ASYM
1180^1 ASYM<1300^2
```

>> TECHNICAL NOTES <<

I. HOW MAB BOOTS

When MAB is BRUN it loads at location 1000 which is where execution begins. The routine at 1000 then relocates MAB between DOS and the DOS buffers. The buffers are then rebuilt. MAB stacks itself below anything already between the DOS buffers (ie. PLE or MAB if it is already there). This makes MAB fairly permanent.

II. MAB MEMORY MAP FOR A 48K APPLE

```
ADDRESS:        USE:

0000..0018      ;UNUSED BY MAB
001A..001D      ;TEMPORARIES USED BY MAB  (Default
                ;may be changed with CONFIGURE.MAB)
001E..0021      ;UNUSED BY MAB
      0022      ;TOGGLED BY 'W' COMMAND ($00<=>$0C)
0023..0039      ;UNUSED BY MAB
003A..003B      ;ASSUMED BY MAB TO CONTAIN THE USER
                ;PROGRAM COUNTER AFTER A BREAK
                ;INSTRUCTION
003C..0044      ;UNUSED BY MAB
0045..0049      ;ASSUMED BY MAB TO CONTAIN THE USER
                ;REGISTERS AFTER A BREAK INSTRUCTION
004A..004B      ;UNUSED BY MAB
004C..004D      ;ASSUMED TO CONTAIN THE INTEGER BASIC
                ;HIMEM ADDRESS
0073..0074      ;ASSUMED TO CONTAIN THE APPLESOFT
                ;HIMEM ADDRESS
0075..00FF      ;UNUSED BY MAB
0100..012F      ;DEFAULT MAB STACK AREA
0130..01FF      ;USER STACK AREA
0200..03EF      ;OPERATING SYSTEM AREA, UNUSED BY MAB
03F0..03F2      ;BREAK JUMP VECTOR USED BY MAB TO
                ;HANDLE BREAK INSTRUCTIONS
```

```
03F3..03F7      ;OPERATING SYSTEM AREA, UNUSED BY MAB
03F8..03FA      ;CONTROL Y JUMP VECTOR SET FOR THE
                ;MAB ENTRY POINT
03FE..07FF      ;OPERATING SYSTEM AREA, UNUSED BY MAB
0800..7000      ;FREE AREA USED BY DOS, APPLESOFT,...
                ;MAB LOADS HERE INITIALLY
7001..76FF      ;THREE DOS BUFFERS (LESS IS SMALLER)
7700..9CFF      ;MAB PROGRAM AFTER RELOCATION.
9D00..BFFF      ;DOS 3.3
C000..CFFF      ;INPUT/OUTPUT AREA
D000..FFFF      ;ROMS OR LANGUAGE CARD
```

III. USE OF MAB CONFIGURE.A

The program MAB CONFIGURE.A allows you to configure MAB or SMAB to use any 4 consecutive bytes of page zero. The default area is 1A through 1D. Another area which is not used is 6 through 9. In any case, you can define any area between 0 and FC (252 decimal) for use by MAB or SMAB. If your program is called by an Applesoft program and does not use Hi-Res graphics,you can also use the graphics registers at $D0-$D5 or $E0-$EA.

This program also allows you to to configure MAB.D000 to use any 1C (28 dec.) consecutive bytes of page zero. The default area is E4 through FF. Since MAB.D000 resides in the Language Card no BASIC may be running at the time so areas used by the BASIC may be used by MAB.D000. To use MAB CONFIGURE.A, run the program and follow the instructions.

IV. REMOVING MAB FROM MEMORY

With MAB now between the DOS buffers, it is virtually impossible to remove without the Control-Z command. Therefore, remember to remove MAB when done. An 'FP' will not be sufficient, as it was on the older version of MAB. If you do get out of MAB and can't remember how to re-activate it with a CALL or Control-Y so as to be able to use the Control-Z to remove it, you'll have to re-boot to restore DOS.

V. MAB AND THE DOUBLE TIME ROM

For those people with the Doubletime Printer F8 ROM installed in their computer, MAB will not function properly unless the IRQ vector is set differently. ($3FE = $43, $3FF = $FA). It is suggested that the EXEC file "DOUBLETIME ROM PATCH" on the MAB diskette be used (i.e. insert MAB disk and type in EXEC DOUBLETIME ROM PATCH), to properly set the IRQ vector for those with DOUBLE TIME machines.

VI. USE OF HANDYSYM.5000

HANDYSYM is a prefabricated symbol table of a few useful labels which may come in handy when creating object code using MAB's mini-assembler. To connect the symbol table, BLOAD HANDYSYM.5000, and then type in K5000. The symbols provided are as follows (subject to change):

```
!Y
FBDD     BELL       {print CHR$(7) character}
5000     COUT       {gen'l purpose output   }
C061     PB0        {loc to read pushbotton #0}
FB1E     PREAD      {read paddles - LDX with paddle #   }
FCA8     WAIT       {wait for value proportional to acc.}
03F5^03  AMPER      {ampersand vector}
03F8^03  CTRLY      {control-Y vector}
03F2^03  RESET      {reset vecor}
C000     KYBD       {keyboard location}
C010     STRB       {kybd strobe location}
C030     SPKR       {speaker location}
FC58     HOME       {clear screen and HOME cursor}
AA60^02  BLEN       {length of last BLOAD}
AA72^02  BADR       {address of last BLOAD}
```

For example, with HANDYSYM.5000 connected, the following operations could be done:

```
!C.HOME              {clears screen}

!A300                {enters assembler}
  300 LDA #"A         {load acc. with "A" character}
  302 JSR .COUT      {print it}
  305 RTS            {routine is done}
  306 <RETURN>       {exit MAB assembler}
```

While MAB and MERLIN are two separate RWP products, it has been anticipated that they will be used in conjunction with one another. The following section deals with their interaction and should help the user to anticipate the few incompatibilities between the two RWP programs.

As mentioned before it is possible to connect a MERLIN symbol table to MAB by simply typing KM from MAB. This works fine, but only if a MERLIN symbol table is present. If the Language Card RAM has been initialized then MAB may act unexpectedly, while an uninitialized Language Card will hang the system.

MERLIN symbol tables are more convenient, however, they are not as versatile as one's created especially for MAB. Specifically, the length associated with the Y command cannot be used with the MERLIN symbol table.

A final note about using MERLIN and MAB in memory together. If the program is a large one that resides in low memory then MERLIN can be Quit and MAB.D000 can be BRUN. This will erase MERLIN on the Language Card and replace it with MAB, but the generated symbol table will remain. This is only valid when the program does't call ROM routines other than in the F800-FFFF area.

It is also possible to have both MERLIN and MAB in memory at the same time. It should be noted that the normal area that MERLIN generates its object code is at $8000 and MAB normally resides in this space. It is therefore first necessary to move MERLIN's HIMEM: down, so that MAB will be protected, and also to ensure that when assembling, object code is generated. The example that follows should be helpful. Just remember where all the pieces sit in memory and be careful that one program doesn't step on another program's space.

To start, BRUN MERLIN, then quit with MAB's "Q" command.

```
Then BRUN MAB

MAB VER 2.6 (C) 1983 WINK SAVILLE
NEXT END
0800 7000
```

Quit MAB with the Quit commend.

```
!Q
TO REENTER MAB -- CALL %30467
```

Write down this number. When entering MERLIN with the ASSEM command the Control-Y vector is rewritten to enter MERLIN. That means this call, or the equivalent from the monitor (7703G), is the only way to get back into MAB.

```
ASSEM
```

This will reenter MERLIN. This can be done directly from MAB but remember to Quit the first time, so the call address to return can be noted.

Once back to MERLIN, go to the EDITOR and type a new value for HIMEM:. This is so MERLIN will generate object code. You can pick any amount of space you want, just remember that the maximum size source file will be smaller. In our case we'll reserve 8K for our object code.

In most circumstances, the end of available memory with both MERLIN and MAB installed will be $8800. Thus to set aside 8K, we should type in:

```
HIMEM: $6800  {from MERLIN}
```

Then load the source file MERLIN SYM EX.S from the MAB disk and assemble it.

```
ASM
                      1    * MERLIN SYMBOL TABLE DEMO
                      2    *
                      3    COUT     EQU  $FDED
                      4    *
6800: A0 00           5    START    LDY  #0
6802: B9 13 68        6    PRMSG    LDA  MESSAGE,Y
6805: 10 06           7             BPL  LASTCHR
6807: 20 ED FD        8             JSR  COUT
680A: C8              9             INY
680B: D0 F5           10            BNE  PRMSG
680D: 09 80           11   LASTCHR  ORA  #$80
680F: 20 ED FD        12            JSR  COUT
6812: 60              13            RTS
                      14   *
6813: D3 D9 CD        15   MESSAGE  ASC  "SYM TEST"
6816: A0 D4 D5 D3 D4
681B: 0D              16            HEX  0D          ;<CR>

--End assembly--

28 bytes

Errors: 0


Now Quit MERLIN

From BASIC type:

CALL 30467  {This should be the call address printed
             by MAB when MAB was quit initially.}

!KM       {Back in MAB connect MERLIN symbol table.}

!Y          {List symbol table}
FDED    COUT
6800    START
6802    PRMSG
680D    LASTCHR
6813    MESSAGE

!-K         {disconnect symbol table}

!Y          {no symbols}
```

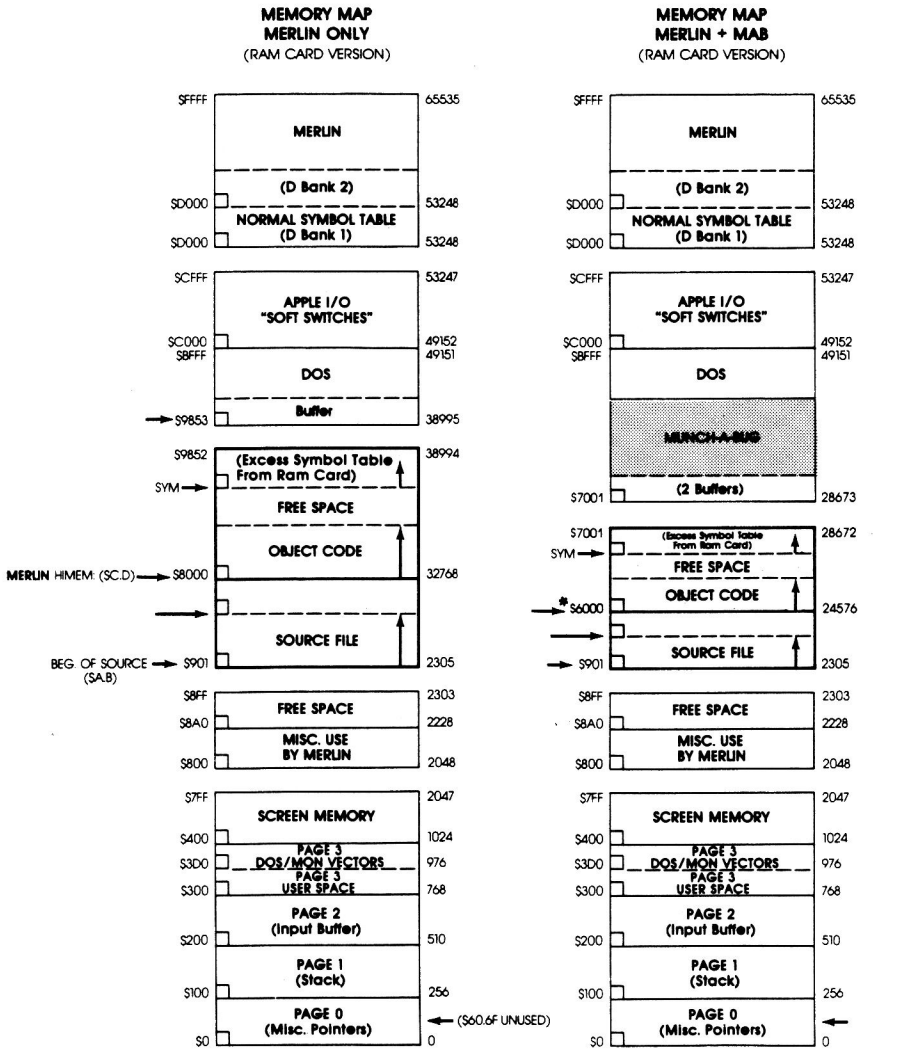
```
!KM          {reconnect symbol table}

!L .START,.MESSAGE      {LIST PROGRAM}
START
 6800  A0 00     LDY   #$00
PRMSG
 6802  B9 13 68  LDA   $6813,Y  MESSAGE
 6805  10 06     BPL   $680D     LASTCHR
 6807  20 ED FD  JSR   $FDED     COUT
 680A  C8        INY
 680B  D0 F5     BNE   $6802     PRMSG
LASTCHR
 680D  09 80     ORA   #$80
 680F  20 ED FD  JSR   $FDED     COUT
 6812  60        RTS
MESSAGE
 6813  D3        ???

!D .MESSAGE,.MESSAGE+9     {DUMP MESSAGE}
MESSAGE
 6813: D3-D9 CD A0 D4 SYM T
 6818: C5 D3 D4 0D FF EST..
```

It is possible to switch back and forth between MERLIN and MAB but be sure to have a back up of the source as it may or may not be necessary to reload the source file each time MERLIN is re-entered. Remember that when things get really fouled up to re-boot and if necessary BRUN LOAD ASM as MERLIN most likely will not reload itself onto the Language Card from a boot or BRUN of MERLIN itself.

The following chart is a comparative memory map of the Apple showing memory allocation with MERLIN alone installed, and with both MERLIN and the standard version of MAB in memory.



*With MAB installed, user must manually reset Merlin's HIMEM:

>> QUICK REFERENCE CARD <<

This chart is intended to provide a quick reference list of MAB commands. The commands are arranged alphabetically and the page number of the manual reference on each command is listed.

Command	Description	Page
A <ae>	ASSEMBLE 6502 MNEMONICS <RETURN> alone to quit. "^" to back up one instr.	38
A 300 A .LABEL A %768		
B ael,ae2,ae3,ae4	BLOCK SEARCH OF MEMORY ael,ae2 = block to search ae3,ae4 = addr. of data to search for.	44
B 1000,2000,300,304 B .LABEL,.LABEL+1000,.BUFFER,.BUFFER+2		
C <ae>	CALL A SUBROUTINE	28
C 300 C.HOME		
D <"or"><ael><,ae2>	DUMP MEMORY	39
D 300 D .LABEL D 300,320		
F ael,ae2,v	FILL MEMORY WITH A VALUE ael,ae2 beg,end of range v = value to fill with.	43
F 300,305,0 F .ENTRY,.END,FF		

```
G ael<,ae2><,ae3><,ae4>  GO WITH BREAKPOINTS        29
                         ael address to start at
                         ae2-ae4 = optional breaks

 G 300
 G 300,302,305,307
 G .ENTRY,.LOOP,.CHK,.END

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

K ae or -K               CONNECT/DELETE SYMBOL TABLE  50

 K 5000
 -K

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

L <ael><,ae2>            DISASSEMBLE MEMORY           12
                         ael,ae2 = start,stop
 L 5000
 L 5000,5050
 L .ENTRY
 L .BEG,.END

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

M ael,ae2,ae3            MOVE MEMORY BLOCK            42
                         ael,ae2 = beg, end of block
                         ae3 = destination
 M 300,305,1000
 M .BEG,.END,.DEST

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

N <ael><,ae2>            NEXT INSTRUCTION <skips JSR's> 21
                         ael = # of instructions
                         ae2 = address to trace
 N 1,1100
 N
 N 3
 N ,1100
-------------------------------------------------------

O <L/H><G/T><M/F><1/2> OTHER SCREEN DISPLAY          49
                         L/H = Lo-Res/Hi-Res
                         G/T = Graphics/Text
                         M/F = Mixed/Full
                         1/2 = Page 1/Page 2
 O HM1
 O T
 O 2
-------------------------------------------------------
```

```
P <ae<,cnt>>        PASS POINTERS                    31
or
P ae @ addr.
                    ae = address of pointer
                    cnt = pass counter
                    @ addr = addr. of check routine
 P 300,1
 P 300
 -P
```

```
Q                   QUIT MAB                         26
```

```
S <ae>              SUBSTITUTE MEMORY                41
                    <"." to exit>
 S 300
 S .LABEL
```

```
T <ae1><,ae2>       TRACE A PROGRAM                  14
                    ae1 = # of instructions
                    ae2 = address to trace
 T 1,1100
 T
 T 3
 T ,1100
```

```
U <ae1><,ae2>       UNTRACE A PROGRAM                20
                    ae1 = # of instructions
                    ae2 = address to trace
 U 1,1100
 U
 U 3
 U ,1100
```

```
V ae1               CONVERT A NUMERIC VALUE          46

 V 300
 V %768
```

```
W <ael>              FREEZE TOP WINDOW                47
                     ael = # of lines to freeze
 W
 W5
```

```
X <register>         EXAMINE A REGISTER               22
 X                   {all registers}
 XA                  {accumulator}
 XX                  {X register}
 XY                  {Y register}
 XP                  {program counter}

 XC                  {column display}
 XD                  {display data= T or F}
 XS                  {stack}
 XM                  {MAB's stack}

Flags: C)arry N)egative Z)ero D)ecimal I)RQ
```

```
Y                    SYMBOL TABLE MANIPULATION        56

Y name,ael<^length or @addr><,ae2>
Y name
Y <^length or @addr.>

 YPTR,6^2,5000       {define PTR=6 with len=2
                      in new table at 5000}
 YPTR                {show value of PTR}
 Y^ or just "^"      {display all non-zero
                      length symbols}
 Y                   {display table names}
```

```
^Z                   PERMANENTLY LEAVE MAB (ZAP)       5
```

MUNCH A BUG™

By Wink Saville

Munch-A-Bug (MAB) is a 6502 program which assists the user in trouble-shooting 6502 assembly language programs by actually running them in a controlled manner. As the program is run, each program step is displayed, along with all 6502 registers, and optional memory points as defined by the user.

This allows you to easily see exactly what occurs at each stage during a program, and quickly find any errors in the code. **MAB** is a professional programming tool, published by Roger Wagner Publishing, Inc., a software company specializing in utility programs for the Apple II.

MAB uses only four zero page locations, and these are definable by the user, so there is never any conflict with a user's program. In addition, **MAB** includes its own mini-assembler, so minor patches to code can be made, right at debugging time, without having to re-run your assembler. **MAB** even supports the use of labels and strings!

MAB also supports such unique features as its "pass pointers". These allow the user to define a point in memory which can be flagged with a set number of "passes". At each pass the 6502 registers will be displayed. On the nth pass, **MAB** stops in the trace mode to further examine the routine in detail. **MAB** also has a "functional pass pointer", wherein a machine language subroutine can be executed at each pass, without affecting the main program (memory of registers). This is useful for performing various tests as to the status of the main program. By setting a special return flag, **MAB** will "pop up" in the trace mode only under certain conditions. These features mean you can easily trace machine language routines linked to Applesoft programs.

Munch-A-Bug's commands include:

- **A)** Assemble 6502 "source code".
- **C)** Call a subroutine.
- **D)** Dump memory in both hex and ASCII.
- **F)** Fill memory with a value.
- **G)** Call a subroutine with breakpoints ("Go").
- **K)** Connect or delete symbol table.
- **L)** Disassemble 6502 code ("List").
- **M)** Move memory.
- **N)** Trace next instruction, but don't display JSR's.
- **O)** "Other" screen display: TEXT, GR or HGR.
- **P)** Display, set, or delete pass pointers.
- **Q)** Quit.
- **S)** Substitute values into a memory range.
- **T)** Trace a 6502 program.
- **U)** "Untrace". Steps through with no display, but with option of keyboard start/stop/end.
- **V)** Convert between Hex and Decimal numbers.
- **X)** Examine and/or modify 6502 registers.
- **Y)** Define, display, or modify symbols (labels).

MAB screen displays are formatted at user discretion to either 40 or 80 column modes. **MAB** can also relocate to any location, including a 16K RAM card, for maximum versatility.

SYSTEM REQUIREMENTS:
48K Apple II, II+, IIe with DOS 3.3

Roger Wagner™ PUBLISHING, INC.

ISBN 0-927796-07-4