DEBUG - DOS/65 DEBUGGER

VERSION 2.1

180 Ridge Road
Cimarron, CO 81220

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION....4

1.1 OVERVIEW AND CONCEPT....4
1.2 KEY FACTS....4

SECTION 2 - EXECUTION....5

SECTION 3 - COMMANDS....6

3.1 OVERVIEW....6
3.2 PROGRAM VARIABLES....6
3.3 BACKGROUND....8
3.4 COMMAND DESCRIPTIONS....9
3.4.1 GROUP 1 (FILE MANIPULATION COMMANDS)....9
3.4.1.1 I (INPUT)....9
3.4.1.2 R (READ)....10
3.4.1.2.1 R WITH NO PARAMETERS....10
3.4.1.2.2 R WITH ONE PARAMETER....10
3.4.2 GROUP 2 (MEMORY EXAMINATION COMMANDS)....11
3.4.2.1 D (DISPLAY)....11
3.4.2.1.1 D WITH NO PARAMETERS....11
3.4.2.1.2 D WITH ONE PARAMETER....11
3.4.2.1.3 D WITH TWO PARAMETERS....11
3.4.2.2 L (LIST)....12
3.4.2.2.1 L WITH NO PARAMETERS....12
3.4.2.2.2 L WITH ONE PARAMETER....12
3.4.2.2.3 L WITH TWO PARAMETERS....12
3.4.3 GROUP 3 (MEMORY MODIFICATION COMMANDS)....12
3.4.3.1 S (SUBSTITUTE)....12
3.4.3.1.1 S WITH NO PARAMETERS....13
3.4.3.1.2 S WITH ONE PARAMETER....13
3.4.3.1.3 S WITH TWO PARAMETERS....13
3.4.3.2 F (FILL)....14
3.4.4 GROUP 4 (CPU STATE COMMANDS)....14
3.4.4.1 X WITH NO PARAMETERS....14
3.4.4.2 X WITH ONE PARAMETER....15
3.4.5 GROUP 5 (EXECUTION COMMAND)....15
3.4.5.1 G WITH NO PARAMETERS....16
3.4.5.2 G WITH ONE PARAMETER....16
3.4.5.3 G WITH TWO PARAMETERS....16
3.4.5.4 G WITH THREE PARAMETERS....16

APPENDIX A - BREAKPOINTS....17

A.1 GENERAL....17

A.2 OPERATION....17
A.3 USER BRK OR INTERRUPT....17
APPENDIX B - DEBUG MEMORY USAGE....20
B.1 OVERLAY CONCEPT....20
B.2 PAGE ZERO USAGE....20
B.3 PAGE ONE USAGE....20
APPENDIX C - INTERRUPTS UNDER DEBUG....23
APPENDIX D - DEBUG DATA LOADING....24

SECTION 1 - INTRODUCTION

1.1 OVERVIEW AND CONCEPT

The DOS/65 Assembly Language Program Debugger, file DEBUG.COM on the distribution diskette, provides the means to load, examine, modify and test programs running under DOS/65. Most of its features can be fully used regardless of the peculiarities of the host system. It is, however, at its best when used in a system which allows modification and use of the 6502 IRQ/BRK vector or a JMP pointed to by that vector. In that case, up to two breakpoints can be used for program debugging. The software is designed to automatically detect whether breakpoints can be used and then reconfigure the software accordingly. Additional detail about the use of breakpoints is contained in Appendix A and a complete description of the available commands is contained in Section 3.

1.2 KEY FACTS

The Version 2.1 DEBUG:

- Includes the ability to view memory as hexadecimal or ASCII data and as machine language.
- Allows DOS/65 files to be loaded, with or without an offset, into the TEA.
- Includes the ability to modify memory.
- Allows execution of programs to be accomplished with zero, one or two breakpoints.
- Will accommodate most common IRQ/BRK and destination JMP ($4C or $6C) combinations.

SECTION 2 - EXECUTION

Program execution is initiated using CCM command line of the general form.

```
(drive:)debug (drive:)(file name.type)
```

The first optional drive field designates from which drive the program DEBUG.COM will be loaded if the default is not to be used. The drive and file name fields following the "debug" are optional. If used they will cause the designated program to be loaded, without offset, into the TEA in preparation for debugging as discussed in Section 3. DEBUG commands are available to accomplish the same effect and in fact those commands allow more flexibility in loading through use of an address offset.

The following are examples of CCM command lines for DEBUG and the resulting action (user inputs are underlined):

COMMAND	ACTION
`A>debug test.kim`	Loads DEBUG.COM from the default drive (A) which in turn loads TEST.KIM into the TEA. TEST.KIM is assumed to be an object code file consisting of "KIM" format records.
`A>b:debug b:test.lib`	Loads DEBUG.COM from drive B. DEBUG.COM loads the file TEST.LIB from drive B. TEST.LIB is assumed to be an object code file consisting of "KIM" format records.
`A>b:debug test.com`	Loads DEBUG.COM from drive B. DEBUG.COM loads the file TEST.COM from the default drive (A). TEST.COM is assumed to be an executable code file beginning at the start of TEA.
`A>debug`	Loads DEBUG.COM from the default drive (A).

SECTION 3 - COMMANDS

3.1 OVERVIEW

The commands available within DEBUG.COM are shown in summary form in Figure 3-1. Those combinations having no entry are not defined and if executed will cause an error message to be generated. Commands are entered by typing the single underlined letter followed by up to three hexadecimal parameters separated by commas. Two commands, Input and Examine, have special command formats. Those formats are discussed in detail in conjunction with the associated command.

3.2 PROGRAM VARIABLES

Understanding of how to use DEBUG.COM depends in part upon knowing what the principal parameters are that DEBUG.COM maintains.

The most important variables maintained by DEBUG.COM are those which describe the host 6502 CPU register and flag state. Upon initial entry to DEBUG.COM those parameters will be set to the following values:

VARIABLE	VALUE
*	TEA
A	0
X	0
Y	0
S	$FF
N	0
V	0
?	0
B	0
D	0
I	1
Z	0
C	0

As indicated, the program counter (*) is set to the value of TEA for which the version of DEBUG.COM being used was assembled. Each register or flag can, however, be altered (see X command) and if breakpoints are used, the registers and flags will reflect the CPU state at the breakpoint location. Conversely, execution of the G command will cause the actual CPU state to be set to the values contained in the variables prior to "jumping" to the address designated by the program counter (*).

The second variable of interest to the user is a pointer into the system memory. It is this pointer which is used during the D (Display), L (List), F (Fill), and S (Substitute) command to view or modify memory. It is not the same as the program counter (*) and thus the D, L, F, and S commands do not alter the program counter and conversely the G (Go) and X (State) commands do not alter the memory pointer.

	NUMBER PARAMETERS			
	NONE	1	2	2
Display	Display bytes @ current pointer	Set pointer to (1) and display bytes	Display bytes from (1) through (2)	
Fill				Fill (1) through (2) with (3)
Go	Execute @ current PC	Set PC to (1) and execute	Set PC to (1), set breakpoint @ (2), and execute	Set PC to (1), set breakpoints @ (2) and (3), and execute
Input	**UFN (filename.type)**			
List	Disassemble instructions @ current pointer	Set pointer to (1) and disassemble instructions	Disassemble instructions from (1) through (2)	
Read	Read file	Read file with offset = (1)		
Substitute	Enter bytes beginning at current pointer	Set pointer to (1) and enter bytes	Set (1) to low byte of (2) and (1) + 1 to high byte of (2)	
eXamine	Display CPU state	**Set designated register or flag to (1) and display CPU state**		
	Bold boxes indicate that command format is peculiar to that instruction. Refer to the command in question for details.			

Figure 3-1 COMMAND SUMMARY

3.3 BACKGROUND

The key features of DEBUG.COM are summarized below.

Prompt

The debugger prompts all command inputs with the character "-".

Parameters

For those commands which use parameters, they are entered in hexadecimal without any prefix or suffix. If multiple parameters are used each parameter is separated from the preceding parameter by a comma. Thus an entry such as

```
D200,2ef
```

will execute the D (Display) command with two parameters. The first parameter will be set to $0200 and the second to $02EF. Leading zeros need not be entered.

If more than four significant hex digits are entered such as in

```
Df2cca
```

the command will not be executed and a "?" will be printed on the console to indicate that an error was detected. Similarly if an illegal number of parameters is entered for the command used, the command will not be executed. If a given parameter is skipped by typing a comma with no other hexadecimal entries, then the parameter is set to 0000.

Upper Case/Lower Case

The debugger includes a case translation feature so that commands, parameters, or file designators may be entered in upper or lower case.

Termination

Control is returned to the DOS/65 CCM by entering a (ctl-c) in response to the DEBUG prompt. A warm boot is executed as a result of this action and the contents of the TEA are not altered. This allows programs or data files to be altered by DEBUG and then saved using the SAVE nn UFN command in CCM.

Editing

Command lines may be edited using normal DOS/65 buffered input editing characters ((ctl-r), (ctl-x), (ctl-p), (delete)).

Output Hold

During long outputs the normal DOS/65 hold character (ctl-s) may be used to freeze the output in order to view a particular portion of the output for extended periods.

3.4 COMMAND DESCRIPTIONS

3.4.1 GROUP 1 (FILE MANIPULATION COMMANDS)

The commands within Group 1 (I and R) provide the ability to load DOS/65 files into the TEA for subsequent modification and testing.

3.4.1.1 I (INPUT)

The I commands allows the user to construct a File Control Block (FCB) for any DOS/65 file. This command does not actually read the file.

The I command is unique in that, while it can have no hexadecimal parameters, it must be followed by a DOS/65 unambiguous file name (UFN). That UFN can also include a drive specification if the desired file is located on a drive other than the default drive. The UFN must meet the DOS/65 standards and thus can not contain illegal characters or spaces and must have a name field of eight or fewer characters and a type field of three or fewer characters. The UFN can be separated from the I (or i) command letter by one or more spaces. The following are examples of legal I commands:

```
i test.com

I a:test.kim

i .b
```

The following are examples of illegal I commands:

`i*`	not a UFN
`i a b.com`	embedded space
`i k:test.kim`	illegal drive
`I testfileO`	UFN too long

3.4.1.2 R (READ)

The R command reads the UFN previously specified by an I command into memory. If the type field of the UFN is COM then the file is assumed to be a machine language file with an implied origin at the beginning of the TEA. For all other types of files DEBUG.COM assumes that the file is an object code file consisting of object code records of the form normally generated by the DOS/65 assembler. While these files are normally of type KIM, DEBUG.COM, treats all files which are not of type COM as "KIM" files. For these files the actual load address is specified by the address field of each record.

3.4.1.2.1 R WITH NO PARAMETERS

If the R command is used without any parameters, COM files are loaded at the start of the TEA. As discussed above all other files are loaded at the address specified by the

address field of each record.

3.4.1.2.2 R WITH ONE PARAMETER
If a single parameter is used with the R command, it is used as an offset for the load address. For COM files, the first load address will thus be TEA + parameter. Since the addition is done as a 16 bit addition and any carry is ignored the effective load address can be anywhere within the 6502 addressing range.

NOTE

Data can be loaded to any location above $200 and below the start of DEBUG.

The following examples illustrate how the offset works for COM files.

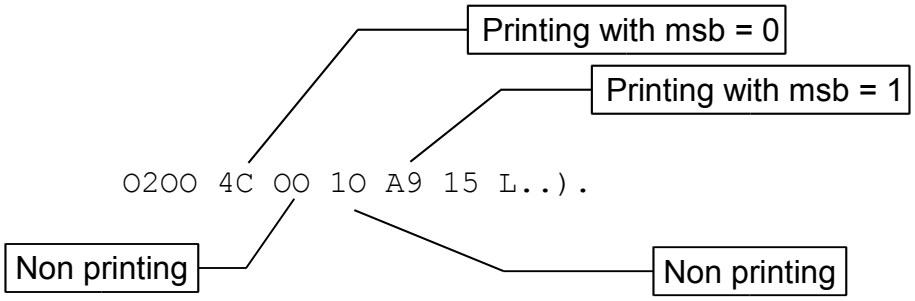
TEA	PARAMETER	LOAD ADDRESS
$2000	E200	$200
$200	1000	$1200
$200	3	$203

For "KIM" files, the effective load address is calculated by adding the offset to the starting address specified in each record. The following examples show how the offset works for "KIM" file.

RECORD	PARAMETER	LOAD ADDRESS
;020200A94100EE	103	$0303
;02F000A02001B1	F000	$E000

3.4.2 GROUP 2 (MEMORY EXAMINATION COMMANDS)
The commands in this group (D and L) allow the contents of memory to be examined either as bytes/characters or as machine instructions.

3.4.2.1 D (DISPLAY)
The D command displays the contents of the specified memory locations as both hexadecimal bytes and, if possible, as ASCII characters. The general form of the display consists of a four digit hexadecimal field showing the first address displayed followed by the memory contents in hexadecimal and then the ASCII characters (msb is ignored) represented by the memory contents. Non-printing characters are shown as the "." character. The number of bytes printed on each line is determined by the console line length specified in SIM. The following line is an example of the D output and shows how ASCII characters are printed.



NOTE

If an attempt is made to execute a D command for which the pointer would "wrap-around" (i.e., go from $FFFF to $OOOO), the command will not be executed.

3.4.2.1.1 D WITH NO PARAMETERS

If no parameters are entered then enough bytes will be displayed to fill or almost fill the console screen. The data will begin at the current pointer. After execution the pointer will point to the byte following the last byte displayed.

3.4.2.1.2 D WITH ONE PARAMETER

This form of the D command functions like the D command with no parameters, however, in this case the pointer is set to the value of the parameter before the command is executed.

3.4.2.1.3 D WITH TWO PARAMETERS

In this case, the pointer is set to the value of the first parameter as was done for the one parameter case. The difference is that the memory contents through the address corresponding to value of the second parameter are displayed rather than only the number required to fill or nearly fill the screen as was the case previously.

3.4.2.2 L (LIST)

The L command lists the contents of the specified memory locations as disassembled machine language. Each line contains one instruction and its operands, if any. All illegal opcodes are printed as question marks. All operands are shown in hexadecimal. The following example shows a typical L command output:

```
0200 4C 00 10  JMP $1000
0203 A9 15     LDA #$15
```

In general the machine language shown is identical to that specified by the DOS/65 Assembler except that the operand field for accumulator mode instructions is blank rather than an A as required by the assembler.

3.4.2.2.1 L WITH NO PARAMETERS
If no parameters are used, the L command will disassemble instructions (not bytes) beginning at the current value of the pointer. Enough instructions will be disassembled to fill the screen. After completion of the operation the pointer will be set to the first byte after the last opcode/operand displayed.

3.4.2.2.2 L WITH ONE PARAMETER
For the L command with one parameter the pointer is set to the value of the parameter and then enough instructions are disassembled to fill the screen.

3.4.2.2.3 L WITH TWO PARAMETERS
Rather than disassembling a fixed number of instructions this mode of the L command will disassemble instructions from the address determined by the first parameter through the address equal to the second parameter. The value of the pointer after completion of this command will be set to the location of the byte immediately after the last instruction disassembled. If the last instruction had no operand then the pointer will be equal to (parameter #2)+1; if one byte operand then (parameter #2)+2; or if a two byte operand then (parameter #2)+3.

3.4.3 GROUP 3 (MEMORY MODIFICATION COMMANDS)
The commands in this group (S and F provide the ability to modify the memory contents on a word, byte, or block basis. As such, they are most useful in entering code or data values during testing of a program.

3.4.3.1 S (SUBSTITUTE)
The S command is the most useful memory modification command. Execution of a S command with zero or one parameter places the system into a special data entry mode. In this mode, the memory pointer and the current contents are printed and the user can modify the contents by entering two hexadecimal digits. When in this data entry mode, all digits must be entered. For example, to enter a value of $00 the correct entry is 00, for $0A it is 0A, and for $11 it is 11. If the current contents are not to be modified, a "." is entered. When modification is complete a (cr) is entered. Any non-hexadecimal character (other than "." or (cr)) will also terminate the S command, however, in that case an error message will be displayed. The following example illustrates how the S command operates. The user entries are underlined.

DISPLAY	BYTE AFTER EXECUTION
-200 4C 20	20
-201 00 .	00
-202 10 F8	F8
-203 50 (cr)	50

Execution of the S command with two parameters functions quite differently and

will be explained below.

3.4.3.1.1 S WITH NO PARAMETERS
For the S command with no parameters execution is as described above. In this case the current value of the pointer is used.

3.4.3.1.2 S WITH ONE PARAMETER
This form of the S commands function as described above except that the pointer is set to the value of the parameter before execution is begun.

3.4.3.1.3 S WITH TWO PARAMETERS
This mode of the S command is a special case which allows both bytes of a word to be modified in one step. The contents of the word are set to the value of the second parameter. The location of the low order byte is the value of parameter 1 and the location of the high order byte is (parameter
1) + 1. Thus the following command

```
S 100, DA06
```

would result in the following:

LOCATION	CONTENTS
0100	06
0101	DA

The chief value of this mode of the S command is in changing JMP or JSR operands in one step. This is especially helpful, for example, if I/O vectors must be altered and if changing one byte and not the other would destroy the users ability to talk to DEBUG.

3.4.3.2 F (FILL)
The Fill command has only one mode and requires three parameters to be entered. The action taken is to fill the memory locations from Parameter 1 through Parameter 2 with the value of Parameter 3. The value of Parameter 3 must be in the range 00 through FF or an error will be detected.

The following command is an example of the Fill command

```
F 200, 20F, 1A
```

In this case the contents of location $200 through $20F will be set to $1A.

At the conclusion of execution the pointer will be set to the value of the second parameter.

NOTE

If the pointer "wraps-around"; i.e., goes from $FFFF to $0000, during execution of a F command, then execution will halt with the last location "filled" being $FFFF.

3.4.4 GROUP 4 (CPU STATE COMMANDS)

The eXamine command is a powerful and useful command for program debugging. It can be used either to display the CPU state or to modify the CPU state. The discussion in Section 3.2 pointed out that DEBUG.COM maintains a set of CPU state registers. The significance of that fact is that upon execution of a G command the CPU state is set equal to the contents of those registers. Similarly, if breakpoints are used then the contents of those registers are set to the value of the associated CPU register or flag at the location of the breakpoint. It is obvious that an ability to view and modify those values is useful in the debugging process.

3.4.4.1 X WITH NO PARAMETERS

In this mode the CPU registers will be displayed in the following format.

```
*    A  X  Y  S  NV?BDIZC

0200 3A 2F 80 FF 10011011
```

The 16 bit program counter (*) is displayed as four hexadecimal digits, the four 8 bit registers (A, X, Y and S) are displayed as two hexadecimal digits, and each of the bits within the processor status byte are displayed as single binary digits.

3.4.4.2 X WITH ONE PARAMETER

The format used to alter a given register or bit is not the same as that used to enter a normal single parameter command. In this case the single parameter must be preceded by the register or bit designator (*, A, X, Y, S, N, V, B, D, I, Z, or C) and an equal sign. Thus the following command

```
xa=3
```

would set A to $03. The parameter must match the specified register or flag in terms of range. The allowable parameter ranges are:

```
*                        0000 through FFFF
A, X, Y, S               00 through FF
N, V, B, D, I, Z, or C   0 or 1
```

If the allowable range is exceeded, an error message will be printed and the command will not be executed.

After the specified register is set to the parameter value, the resulting CPU state is printed

as is done for the X command with no parameters.

3.4.5 GROUP 5 (EXECUTION COMMAND)

The single execution command available within DEBUG is the G command. This command sets the CPU registers to the current values of the CPU State registers. Variations of this command also allow the program counter (*) to be altered prior to execution. Even more importantly, variations are available so that the user can set one or two breakpoints. If the breakpoints are encountered during execution of the program in memory, the user is returned to the DEBUG command mode and the instruction at which the break occurred is shown in disassembled form and the CPU state when the break was encountered is displayed. The following example illustrates a typical result of a G command. (Command format is explained in the following sections.)

```
-G 200,3FF,101F
101F      20 1A 32  JSR $321A       Display After
*    A  X  Y  S  NV?BDIZC           Breakpoint @
101F 5A 33 8F FC 10010101          $101F Is Encountered
```

NOTE

All breakpoints are cleared upon return to DEBUG and execution may be continued by execution of another G command. If the user is not satisfied with the results, the CPU state can also be altered using the X command before program execution is continued.

3.4.5.1 G WITH NO PARAMETERS

This mode of the G command will cause execution of the program in memory to be initiated (or continue) at the current value of the program counter (*). No breakpoints are set and all CPU registers are set to the contents of the CPU state registers.

3.4.5.2 G WITH ONE PARAMETER

With one parameter the G command is executed without breakpoints. The program counter is set to the value of the parameter and all other CPU registers are set to the values of the CPU state registers.

3.4.5.3 G WITH TWO PARAMETERS

For this mode of the G command a single breakpoint is set at the location determined by the second parameter. The program counter is set to the value of the first parameter except that if the value of the first parameter is zero, the current program counter is used without alteration. This allows rapid continuation after a previous breakpoint by entering a command such as

```
G,AC1B
```

which will execute at the current program counter and will set a new breakpoint at $AC1B.

3.4.5.4 G WITH THREE PARAMETERS

This mode of the G command functions just like the G command with two parameters except that two breakpoints are set. The first breakpoint will be set at the second parameter and the second breakpoint will be set at the third parameter.

APPENDIX A - BREAKPOINTS

A.1 GENERAL

The use of breakpoints in program debugging for the 6502 is facilitated by the BRK opcode ($00) and the IRQ/BRK vector located at $FFFE. In most systems that vector is either in RAM or points to a JMP ($4C) in RAM or to an indirect JMP ($6C) for which indirect address in contained in RAM. DEBUG searches for one of those conditions at initial execution. If DEBUG finds one of those conditions it sets a flag indicating that breakpoints are allowed. At the same time, the vector or jump is set to point to a special routine in DEBUG and an appropriate message is sent to the console. If one of those conditions is not found, breakpoints are not allowed and the user is so informed. Since breakpoints can not be set in this case, the G commands which execute and set breakpoints will cause execution to occur but without breakpoints.

CAUTION

It is possible for DEBUG to enter an infinite loop if the IRQ/BRK vector is in ROM and points to a JMP ( ) for which the indirect address is in ROM. If, for example, the IRQ/BRK vector was in ROM and was set at $F800 and if $F800 contained a JMP ($FFFE), i.e., a $6C $FE $FF, then DEBUG would loop forever.

A.2 OPERATION

The functional flow which occurs after a G command is executed with breakpoints is illustrated in the pseudo-code at the end of this appendix. Note that once the user program is entered the only way that the user can normally force termination of the program is through use of such steps as a system RESET or a NMI if the system allows such action. If either of these two steps is taken it is possible that DOS/65 can be re-entered without destroying the program in memory if the user has available a monitor which can be used to execute at designated locations and if the RESET or NMI does not destroy the contents of memory. Probably the most obvious way of doing that is to go (using the monitor) to $100 and execute that JMP. That action will cause a warm boot to occur after which the memory contents can be saved. DEBUG could then be re-executed using the saved program. This multi-step process is essential since DEBUG initially overlays the lower part of the TEA when executed. (See Appendix B)

A.3 USER BRK OR INTERRUPT

DEBUG does not preclude use of interrupts or the BRK command by the user. Although DEBUG sets the IRQ/BRK vector (or the destination JMP) to point to the BRK handler in DEBUG, the original destination address is preserved by DEBUG. When the handler is entered due to an interrupt or a BRK, it determines whether the action was the result of a breakpoint set by DEBUG or some other action. If the cause was not due to a DEBUG set breakpoint then control is passed to the address originally set by the user in the IRQ/BRK vector (or the destination JMP). One limitation of the approach used to allow this flexibility is that the execution time of an interrupt or user BRK when running under DEBUG is

significantly slower. If IRQ or BRK response time is critical, it is most likely that breakpoints should not be allowed. As discussed above DEBUG will not alter the IRQ/BRK vector (or destination JMP) if it detects that the vector is in ROM and if the destination pointed to by the IRQ/BRK vector is not a JMP.

The key to "tricking" DEBUG then is to ensure that one of the latter conditions exists when DEBUG is executed. Making the destination of the IRQ/BRK vector something other than a JMP is probably the best since even if it should be a JMP for correct operation it is easy to change it to the correct opcode (using DEBUG) before the program is executed.

PSEUDO-CODE

FLOW AFTER G COMMAND EXECUTION

```
IF PARAMETER 1 NOT ZERO
        THEN
                SET * TO PARAMETER 1
IF BREAKPOINTS ALLOWED
        THEN
                INSERT BRK AT PARAMETER 2 AND PARAMETER 3
SET CPU FROM (* A X Y S N V B D I Z C)
JMP *
```

USER PROGRAM

```
START AT *
IF BRK OR IRQ
        THEN
                JMP ($FFFE)
```

FLOW UPON BRK OR IRQ RETURN TO DEBUG

```
IF USER BRK OR IRQ
        THEN
                JMP USER IRQ/BRK
CLEAR BREAKPOINTS
SET POINTR TO *
EXECUTE L AT *
EXECUTE X
WAIT FOR USER COMMAND
```

APPENDIX B - DEBUG MEMORY USAGE

B.1 OVERLAY CONCEPT

DEBUG is a unique program since it is a transient which allows other transients to also be executed. This capability is realized by having DEBUG relocate itself after being loaded but before the program to be debugged is loaded. Figure B-1 shows what the system memory looks like immediately after DEBUG is loaded and what it looks like after DEBUG is relocated.

The most important aspects of this are:

- The process of loading DEBUG destroys the previous contents of the lower part of the TEA.

- DEBUG during execution replaces the CCM.

- The JMP to PEM at $103 is modified to point to a JMP located at the beginning of DEBUG. This allows user programs which use the vector at $103 to determine how much memory is free to function without destroying DEBUG.

CAUTION

Free user memory will be reduced when DEBUG is running and thus there may be insufficient memory for some applications.

B.2 PAGE ZERO USAGE

DEBUG does use the first few bytes in page zero immediately after being loaded to accomplish the relocation. Once DEBUG is relocated and executed it does not use page zero memory.

B.3 PAGE ONE USAGE

DEBUG uses the top few bytes of Page One for the stack and uses the lower portions of Page One for the normal DOS/65 JMP's, FCB, and buffer.

	SIM	SIM
	PEM	PEM
	CCM	EXECUTE MODULE
		JMP PEM
	EXECUTE MODULE	
TEA START >	LOAD MODULE	
$103 >	JMP PEM	JMP EXECUTE-3
$100 >	JMP SIM+3	JMP SIM+3
	AS LOADED	EXECUTING

DEBUG MEMORY MAP

APPENDIX C - INTERRUPTS UNDER DEBUG

Problems may be evident when using DEBUG.COM with programs which alter the IRQ/BRK vector or the JMP pointed to by that vector. At initial execution DEBUG.COM alters either the vector or the JMP so that breakpoints can be used. Because of that, a program which is run under DEBUG can not then alter the vector or JMP. If either is altered, strange things will happen when trying to run DEBUG with breakpoints.

There is a way around the problem. DEBUG does preserve whatever was in the vector or JMP when it is executed and then will allow a user interrupt routine to be executed after determining that the interrupt was not due to a DEBUG set breakpoint. The trick then is simply to make sure that the vector or JMP is set to the correct location before executing DEBUG. A short program could be written to do that or the program to be debugged could be used with an appropriate dummy exit inserted to kill execution after the vector or JMP is set. When the program to be debugged is executed under DEBUG, the code which sets the vector or JMP should be bypassed.

APPENDIX D - DEBUG DATA LOADING

DEBUG prevents the user from loading a .COM or .KIM file to any location at or above the start of the DEBUG execute module. While a reasonable self protection feature, it is restrictive. Some users may have memory above the region normally occupied by transients or DOS/65 and would like to be able to load files into that region. The following modification to DEBUG can be used to defeat the checks which prevent loading data at or above the start of DEBUG.

CAUTION

With the following modification in place it will be easy to crash the system. Use great care in loading files with the modified DEBUG. Values shown below are valid for DEBUG.COM V2.02 only.

Step 1 Enter the following CCM command:

```
DEBUG DEBUG.COM
```

Step 2 Use the S command to change the following bytes to NOP ($EA).

```
TEA+$588
TEA+$589
TEA+$58A

TEA+$62D
TEA+$62E
TEA+$62F
```

Step 3 Enter ctl-c.

Step 4 Enter the following CCM command:

```
SAVE 14 XDEBUG.COM
```

It is recommended that the standard DEBUG be retained and that this modified version only be used when necessary to minimize the risk of crashes.