

**DEBUG for the TANDY Model 100/102**  
**Version: 4.13 14/June/1990**  
**Copyright: 1986,1987,1988,1990 Adrian Ryan**

**DISCLAIMER**

This software is provide to you on an “as-is” basis. You are entirely responsible for its use. The author makes no warranties as to the suitability, functioning, or correctness, nor accepts any liability for loss or damage however caused.

This is not intended to convey the impression that this software will not be supported, rather, since it is a powerful tool whose use is entirely outside of my control, I wish it to be clearly understood that since DEBUG can operate upon any memory location, any I/O port, etc, it is quite possible for you to wipe out files and lock up the machine. If all else fails, turn all the power OFF, including the memory power and start again.

**COPYRIGHT**

This product has been developed and refined over a number of years, and I am constantly honing and improving it. This version is distributed to you as shareware. You are encouraged to distribute this program and documentation providing:

- [1] ALL the files are distributed;
- [2] Neither the program nor the documentation files are modified in any way;
- [3] This copyright notice, and that appearing in the program are not removed or altered.
- [4] No charge is made, other than a media replication fee.

**CONTENTS**

This package consists of the following files:

- [1] DBMAKE.DO a BASIC program to re-generate the file DEBUG.CO from the data file DBHEX.DO
- [2] DBHEX.DO ASCII hex data file for item [1]
- [3] DBDOC.DO this document.
- [4] DBCFG.DO a BASIC program to re-locate the DEBUG.CO file to any desired address.

**INITIAL SETUP**

In order to re-construct the machine program DEBUG.CO and save it as a file you will need at least 21K of free memory. This is only necessary the first time you perform this operation, DEBUG itself is only 4.9K. Clear memory of all other programs, and load the files DBHEX.DO and DBMAKE.DO. Enter BASIC and execute the command:

```
CLEAR 512,58040
```

Now load and execute the file DBMAKE.DO. The program will check that sufficient memory exists to re-construct DEBUG, and will abort with a message if this is not the case. If all is OK, then the program will prompt you for the name of the data file. Enter the name you used, normally DBHEX.

The program will then display the address and the data as it re-constructs the machine code from the hexadecimal data file. At the end of the process the checksum is compared. If no errors are encountered the amount of free memory remaining will be checked, and if you have sufficient then you will be prompted whether you wish to save DEBUG as a file.

Once you have the machine code version of DEBUG, you can save it to cassette or disc, and you will not need to use either the DBHEX.DO or the DBMAKE.DO files again.

### CONFIGURATION

DEBUG can be configured to run at almost any memory location with the aid of the utility package DBCFG.DO. Initially, DEBUG.CO is assembled to load into memory from address 58040 through to 62959, with an entry point at 58040.

If you do not have any other machine language programs that use this address space, DEBUG can run at this location. However, if you have additional software, for example the Disk Operating System for the TANDY DVI, or Floppy.CO, or the RAM resident version of TS-DOS for the portable disk drive, then DEBUG must be re-located to avoid conflict with these programs.

In order to perform this re-location, load the DBCFG program. Enter BASIC and clear memory (save any important files first) with the command:

```
CLEAR 0,58040
```

Then, if DEBUG.CO is not already saved as a RAM memory file, load DEBUG with the command:

```
CLOADM "DEBUG" {Assumes DEBUG.CO is on cassette}
```

The machine will load the DEBUG.CO file and should show the following display:

```
Top: 58040
End: 62959
Exe: 58040
```

Save the file with the command:

```
SAVEM "DEBUG" 58040,62959,58040
```

DEBUG needs 4919 bytes of memory in order to run, and thus the highest address at which you can load it is 4920 bytes lower than the TOP address of any program with which it must co-exist. Set the HIMEM pointer for this co-resident software, and load it into memory. Now enter BASIC and enter the instruction:

PRINT HIMEM - 4920

The value printed is the highest address at which you can load DEBUG in order to co-exist with the existing software. DBCFG will examine the directory looking for DEBUG.CO. If it is missing, or if the wrong version of DEBUG is found, DBCFG will display an error message and abort.

Assuming that the correct version is found, you will be prompted with the current address information of DEBUG.CO and queried for a new starting address. Enter the value that was calculated earlier. You will be asked if you wish to proceed.

If you answer anything other than 'Y' or 'y', DBCFG will exit without making any changes. If you continue, then the display will blank for a few seconds, then the message:

Address Offset: xxxx

will be displayed, where 'xxxx' will be a rapidly changing number. At the end of the process, DEBUG will have been modified in its directory file slot to exist at the new target location. Load and execute as for any other machine language program.

Note that the re-location process is reversible. DEBUG.CO can be re-located as many times as you wish. In order for this to occur correctly, it is VITAL that neither DEBUG.CO nor DBCFG.DO are modified in any way! Failure to heed this warning will result in severe corruption of DEBUG, with unpredictable results.

## DEBUG COMMAND REFERENCE

[A]ssemble addr1  
[B]reakpoint [addr1]  
[C]alculate arg1 [arg2]  
[D]isplay addr1 [addr2]  
[E]nter addr1 arglist  
[F]ill addr1 addr2 num/char  
[G]o addr1  
[L]ength [num]  
[M]ove addr1 addr2 numbytes  
[P]rinter  
[Q]uit  
[R]egister [arglist]  
[S]earch addr1 addr2 arglist  
[U]nassemble addr1 [addr2]  
[V]erify addr1 addr2 numbytes  
[W]alk addr1

### ASSEMBLE

Syntax : A addr1  
Example : A C000

This command is a line oriented assembler for the 8085, which will start the assembly at addr1. The standard mnemonics are used, with a minor restriction.

In order to minimise the amount of space occupied by the mnemonic op-code table, the op-codes PUSH/POP PSW have been changed to PUSH/POP AF.

All numeric data must be entered as pairs of HEXADECIMAL digits, and must be preceded by the \$ symbol. Note however, that the RST command operands are considered part of the op-code, and therefore are entered as a single digit without the \$ prefix. Thus, for example:

```
MVI A,$06  
LXI H,$1234  
JMP $C000
```

but:

```
RST 4
```

A legal statement can consist of any number of leading spaces, a 2, 3, or 4 character mnemonic op-code, a legal separator, and the legal operand field for the chosen mnemonic. Legal separator characters are the tab, comma, and space symbols.

Thus the following lines are all legal:

```
MVI A $FF
MVI A,$FF
MVI ,A,$FF
```

or any combination of the above. All lower case entries are converted to upper case prior to being processed.

The command will terminate when a blank line is entered. Illegal mnemonics/operations will cause a beep, and will be ignored.

When the ENTER key is pressed, if the line was acceptable, it is immediately assembled and entered into memory. Then the dis-assembler routine will examine the code and display exactly what was entered so that you may verify that your intentions have been correctly interpreted.

See GENERAL NOTE [7] for illegal op-code recognition.

### **BREAKPOINT**

Syntax : B [addr1]  
Example : B C025

This command will set or clear a breakpoint at the specified address. When a breakpoint is encountered, the original code at that address will be restored, and the program will halt and display the dis-assembled code at the breakpoint, as well as the current register contents.

If the address specification is omitted, then the breakpoint will be cleared if it was set, or if it was not set, the command will be ignored. Note that the WALK and QUIT commands will restore any set breakpoints before executing. Furthermore, breakpoints may only be set in RAM, and if an attempt is made to specify a ROM address, the routine will beep and return to the command line.

### **CALCULATE**

Syntax : C arg1 [arg2]  
Example : C 0034 FE11

This command, if followed by two parameters, will calculate their sum and difference, if only a single argument is specified, the ASCII 2-byte equivalent will be displayed, along with the binary and decimal equivalents.

### **DISPLAY**

Syntax : D addr1 [addr2]  
Example : D C000 C123

This command displays the contents of memory from addr1 to addr2 in HEX and ASCII, either on the printer or on the current video screen. If only one argument is supplied, then the 8 bytes at that address will be displayed, and the program will wait. If the cursor up key is pressed, the next 8 bytes of memory will be displayed, if the cursor down key is pressed, the previous 8 bytes will be displayed. The ENTER key terminates the command.

If both address arguments are given, a continuous memory dump occurs, in ascending address order, on the currently selected output device. (Screen or Printer) If the device is the screen, the display will pause after a certain number of lines have been displayed, and will continue when the SPACE bar is pressed. The printer is assumed to be a continuous display device, and no pause is made.

Whilst a continuous dump is in progress, hitting any alphanumeric key will temporarily suspend the output. Hitting the SPACE bar will cause the dump to resume. This facility is applicable to both screen and printer. See the P and L commands.

## **ENTER**

Syntax : E addr1 arglist

Example : E C000 AB F0 'abcdef' 0D 0A

This command will permit entering HEX or ASCII characters into memory from addr1. To specify that the characters are ASCII, the string must be enclosed in the single quote character, ('). The arglist may be composed of any combination of HEX digit pairs or delimited ASCII strings up to a maximum composite string length of 64 characters. Although for clarity spaces can be placed between the HEX digit pairs, they are not necessary.

Thus, the commands:

```
E C000 FF AA 'TRS-80 Model 100' 00
```

and:

```
EC000FFAA'TRS-80 Model 100'00
```

are equivalent.

Similarly, the commands:

```
E C000 0A 0D 'TRS-80' 20 'Model' 20 '100' 00
```

and:

```
EC0000A0D5452532D3830204D6F64656C2031303000
```

are identical as far as the ENTER command is concerned.

## **FILL**

Syntax : F addr1 addr2 num/char

Example : F C000 CFFF E0

This command will fill the memory from addr1 to addr2 with the value of num, or the ASCII character char. An ASCII character is specified by preceding it with the single quote character, (').

## **GO**

Syntax : G addr1  
Example : G C000

The code pointed to by addr1 will be executed by means of a CALL instruction, using the current pseudo-register contents. To ensure that control is returned to DEBUG, your code must end with a RET instruction. When control returns, the register contents are displayed, and DEBUG will restore the stack pointer to its default setting.

See GENERAL NOTE [3] regarding use of stack.

## **LENGTH**

Syntax : L [num]  
Example : L 05

This command sets the number of lines on the active display screen. In the case of the LCD, the maximum practical number is 8, although the command will accept all values from 1 to 255. The effect is to pause the current display after num lines have been displayed. Pressing the SPACE key will continue the display for a further num lines. This command has no effect on the printer. If the optional argument is omitted, the current length setting is displayed.

## **MOVE**

Syntax : M addr1 addr2 numbytes  
Example : M C000 E000 01F4

The memory contents pointed to by addr1 will be moved to addr2 for the number of bytes specified by numbytes. Overlapping source and destinations are permitted.

## **PRINTER**

Syntax : P

This command will alternately toggle the output of the D, S, U and V commands between the current display device and the printer. If a printer is not connected or is busy, the program will wait indefinitely. The L command has no effect on the length of the printer output.

## **QUIT**

Syntax : Q

This command will quit DEBUG and return you to the MENU, restoring any set breakpoints in the process.

## REGISTER

Syntax : R [arglist]

Example : R 00E3 FFFF 1234 A000

This command displays the pseudo-register contents if there is no arglist, or, if the optional arglist is present, will set the pseudo-registers to the values specified and then display the contents.

Registers are specified in the order:

AF BC DE HL

A comma must be used to skip over a register. Thus, for example, if you wish to set the HL register pair to C000 without disturbing the other registers the command would be:

R , , , C000

Should you wish to set only the AF register, for example, then the command:

R E800

would set the A register to the value E8, and the F register to 00, leaving the other register settings undisturbed.

See GENERAL NOTE [8] regarding illegal flags.

## SEARCH

Syntax : S addr1 addr2 string

Example : S C000 E000 'abcdef' AD BC 00

This command will search memory from addr1 to addr2 examining each byte trying to match the string of ASCII characters or HEX digits specified in the third argument. In the example given, the search will be from C000 to E000 for the text string abcdef followed by the HEX digits AD BC 00.

To specify that the characters are ASCII, they must be enclosed by the single quote (') delimiter. Both delimiters must be present! Any other characters will be treated as pairs of HEX digits.

**Note:** Be careful about the single quote character! The only acceptable form is the forward sloping quote – if you are using this software with, for example the Virtual-T emulator, the normal PC keyboard has both a forward and a backward sloping quote character.

If an illegal character is specified, the command line will be ignored. The search will find all occurrences of the target string, and the addresses displayed correspond to the location of the first character. Only the command line prompt will be displayed if the target string is not found. Note that for a search operation to be successful, an exact match must occur. Upper and lower case letters in a delimited ASCII string do not match. The maximum search string length is 64 bytes.



The output of this command may be re-directed to the printer with the 'P' command, and can be suspended during execution by hitting any alpha-numeric key. Output can be resumed by hitting the SPACE bar. The execution can be aborted by hitting the ENTER key.

### **UN-ASSEMBLE**

Syntax : U addr1 [addr2]

Example : U C000 C123

This command will dis-assemble the code from addr1 to addr2 and display the results on the current screen. The mnemonics are consistent with those used by the ASSEMBLE command.

If the second argument is omitted, then only one line of code will be unassembled. The program will then wait for either the cursor up key to be pressed, which will cause the next line to be displayed, or the ENTER key, which will terminate the unassembly.

The same SUSPEND, LENGTH, and ABORT facilities are available as for the D, S, and V commands.

See GENERAL NOTE [7] for illegal op-code recognition.

### **VERIFY**

Syntax : V addr1 addr2 numbytes

Example : V C000 D000 1000

This command will compare the contents of memory between addr1 and addr1 + numbytes with the contents of memory from addr2 to addr2 + numbytes. If no differences are found, then only the command line prompt will be returned.

If a difference is found the address of the first section and the contents as well as the corresponding address/contents of the second section will be displayed. This command will also allow output to be diverted to the printer, and will pause at the end of the currently defined screen, as well as responding to the ENTER key to abort the output.

### **WALK**

Syntax : W addr1

Example : W C000

This command permits single stepping the code starting at addr1. Prior to executing the next instruction, DEBUG will display the current register contents and dis-assemble the instruction. WALK has four sub-commands:

#### **Single Step**

Pressing the cursor up-arrow key will cause DEBUG to execute the currently displayed instruction. At the end of the execution, the new register contents will be displayed, and the next instruction dis-assembled.

### **Execute Sub-Routine**

Pressing the 'X' key will allow you to execute a sub-routine CALL/RST at full speed and return control to DEBUG. This facility saves you having to single step through code in a sub-routine that is already well behaved, for example, the ROM routines. Note that DEBUG will check to verify that the code byte is actually that of a CALL, RST or a conditional CALL. All other op-codes are ignored.

### **Execute Code**

Pressing the 'G' key will cause DEBUG to transfer control to the code at the current address. It is exactly the same as exiting from WALK and entering the GO command using the current address. As with the GO command, in order for control to be returned to DEBUG, your code must end with a RET or conditional RET op-code.

### **Quit**

Depressing the ENTER key will terminate the WALK command without executing the currently displayed code. DEBUG will restore the default stack pointer setting on exit.

Note that both RAM and ROM code can be walked.

See GENERAL NOTE [7] for illegal op-code recognition.

### **XPAND FLAGS**

Syntax : X [arglist]

Example : X E3

This command will display the current flag register contents in an expanded form. If the command is followed with an argument then the flag settings corresponding to this HEX byte value will be displayed. The existing flag register settings will not be altered. See REGISTER.

See GENERAL NOTE [8] regarding illegal flags.

### **GENERAL NOTES**

#### **[1] Multiple Commands**

Multiple commands may be combined on the same line. For example, to set the registers to a particular value and to execute the code at a certain address the command line:

```
R FFE3 F123 1234 FFFF:G C000
```

can be executed. The maximum command line length is 254 characters, and is stored in the input buffer starting at \$F685.

Each command must be separated by the colon (:) character, and spaces are ignored except when part of a delimited ASCII string. When a sequence of commands is given, if DEBUG detects any errors whilst either parsing the command string, or attempting to execute the command, then it will abort without executing any following commands. If a major error is discovered, then DEBUG will complain with a two-tone beep, whilst if a minor error was discovered, only a

single tone beep will occur. A major error, for example, would be attempting to execute a command that is not implemented. A minor error would be the presence of invalid characters in an argument, or an illegal address.

Note that the 'A', 'G' and 'W' commands are considered to be self-terminating, that is, they should be the last command on a multiple command line.

## [2] Case/Range Sensitivity

Commands may be entered in upper or lower case. All commands and command line HEX arguments are converted to upper case internally before processing.

For any commands that can have an address range specification, note that it is immaterial in which order the address references are given. For the D and U commands the range will be from the lowest to the highest address, whilst the F, S, and V commands will always perform their tasks from the highest location downward.

## [3] DEBUG Stack

DEBUG maintains its own stack, which is set to 256 bytes, and is located immediately in front of the main code. Since the code that is being debugged may mutilate the stack, the actual initial stack pointer setting is 32 bytes lower than the top of the stack, thus your code can mis-manage the stack for at least 16 pops too many before any corruption of DEBUG occurs.

Every time either the WALK or the GO commands terminate, after saving the current register contents in the pseudo-registers, the stack pointer is restored to its initial setting 32 bytes lower than the end of the stack space.

If the code that you are examining uses its own local stack, then prior to using either the GO command or the WALK 'G' sub-command you must ensure that the return address to DEBUG is preserved at the top of the new local stack.

A section of code thus:

```
SHLD  TEMP          ;Save current contents of HL
POP   H              ;Recover return address to DEBUG
SHLD  RETADDR        ;Save return address
LHLD  TEMP           ;Restore old contents of HL

{your code section goes here}

SHLD  TEMP          ;Save current contents of HL
LHLD  RETADDR        ;Recover DEBUG return address
PUSH  H              ;and put on top of stack.
LHLD  TEMP           ;Recover old HL contents,
RET                    ;and return to DEBUG.
```

will ensure that you will return to DEBUG. Note that only the 'G' command suffers from this restriction, the 'W' command is not affected by any local user change of stack, except insofar as the user stack must have enough room to accommodate DEBUG's requirements. Typically, if you allow at least 72 bytes for DEBUG, then there should not be any stack over-run problem.

Note also that on return from a 'W' or a 'G' command DEBUG will restore the original default stack that was used on entry to the program.

#### [4] Pseudo-Register Contents

The contents of the pseudo-PC are only updated during the WALK command, and when a breakpoint is encountered. This register is not restored to the machine when a GO/WALK command is executed, for obvious reasons. Thus the value displayed in this register represents the place in the code at which either the WALK was terminated, or at which a breakpoint was encountered.

The Pseudo-SP register correctly reflects either the current contents of the SP register during a WALK or the last contents of the SP register prior to exiting from either the 'W' or 'G' commands.

#### [5] Cautions using the E/F Commands

Exercise caution when using the ENTER/FILL commands. Even if DEBUG is entered from the main menu, the operating system maintains a stack somewhere in memory, and an inadvertent alteration of this memory with either the E/F commands will cause unpredictable results. In general, the computer will execute a cold start if you accidentally overwrite a vital location.

#### [6] ROM Usage

DEBUG makes considerable use of the Model 100 internal ROM routines, both the documented calls and various undocumented routines. The code has been tested on a Model 100 using the TANDY DVI, with no problems, as well as the Model 102.

#### [7] Illegal Op-Code Recognition and Usage

This revision of DEBUG now recognises 9 illegal op-codes:

\$08/DSUB B, \$10/RHR, \$18/RDL, \$28/DMOV H,\$nn \$38/DMOV S,\$nn \$D9/SHLX,  
\$DD/JND \$nnnn, \$ED/LHLX, \$FD/JD \$nnnn.

The code represented by HEX \$CB is still regarded as illegal since it is useless on an M100. Note however that these codes will only execute correctly if your 80C85 behaves like most others. You must test these codes yourself on your machine.

The ASSEMBLE/DIS-ASSEMBLE and WALK routines will recognise these codes and act appropriately.

**Note:** DEBUG itself makes no use of these illegal op-codes. However, it is obvious that some commercial software does use them, and even Microsoft uses them in the ROM, thus despite my inherent dislike and distrust of these "features", I felt obligated to include them in this tool.

\$08	DSUB B	[HL] = [HL] - [BC]
\$10	RHR	[HL] = [HL->] HL arithmetic right shift
\$18	RDL	Rotate DE left through Carry
\$28	DMOV H,\$nn	[DE] = [HL] + \$nn (unsigned)
\$38	DMOV S,\$nn	[DE] = [SP] + \$nn (unsigned)

\$CB	??? Invalid.	CALL \$0040 if V flag set
\$D9	SHLX	Store [HL] at address pointed to by [DE]
\$DD	JND \$nnnn	Jump to \$nnnn if flag bit 5 is clear
\$ED	LHLX	Load [HL] from address pointed to by [DE]
\$FD	JD \$nnnn	Jump to \$nnnn if flag bit 5 is set

## [8] Illegal Flags

In this version of DEBUG the register display will now reflect the state of the two “unmarked” flag bits. To remind you that they are not “legal”, they are displayed in lower case, thus a typical flag register display will be:

```
SZdA PvC
10110101
```

where the ‘v’ flag, bit 1, is a conventional overflow flag, that is, it is set when the carry into the sign bit does not equal the carry out. It signifies that the result of an operation on two signed bytes was too big to fit into one signed byte. The ‘d’ flag, bit 5, is a “true” sign bit. It is what would be in bit 8 if the CPU were doing 9 bit signed arithmetic.

Document Revision Date : 7 March 1989  
 DEBUG Version 4.13 14 June 1990  
 PDF Version : 11 June 2013