

DEBUG TOOLS

This document describes the tools and utilities needed to successfully install, configure and re-configure DEBUG for the Tandy Model 100.

DEBUG is a machine-language tool written specifically for the Tandy Model 100/102 and Olivetti M-10 notebook computers. It provides the machine language programmer with the facilities to inspect, modify, assemble, dis-assemble, walk, run, move and verify code in the target machine.

Normally you will receive the following installation files:

DEBUG.DO	ASCII HEX text file of the program
DBMAK.DO	ASCII BASIC text file of the installation program
DBCFG.DO	ASCII BASIC text file of the re-locator program
HEXDAT.BAS	GW-BASIC ASCII text file to convert the data file
MAKEDATA.BAS	GW-BASIC ASCII text file to data statements
DBUG.DO	ASCII text file of the DEBUG reconstructor program
DEBUGH.HEX	ASCII object file at original address
DEBUGL.HEX	ASCII object file origin 256 bytes lower
DEBUG413.ASM	Assembly language source file
XASM85.COM	8080/8085 Cross assembler
BASIC.EXE	GW-BASIC Interpreter

There may also be various documentation files, including an Adobe PDF file containing the installation and usage instructions.

This package contains the development files used to produce the above files, and is useful if you wish to modify the program to suit your specific needs.

The first file is DEBUG413.ASM, the assembly language source file. Before attempting any modifications, it would be wise to read the entire file to gain an understanding of how it works.

This file is assembled using the MS-DOS tool XASM85.COM. Bearing in mind that these tools were developed for use in a now obsolete operating system, it is not surprising that they do not run well on modern systems.

In particular, since the assembler is a COM file, which was obsolete even in the days of MS-DOS, it will not run on a modern machine if the operating system is later than Windows 95. The DOS compatibility mode of later operating systems do not fully emulate the original DOS environment, and certain COM files simply will not run.

The COM file format is basically a memory image, and was a hangover from an even earlier operating system CPM. Indeed, MS-DOS preserved the original CPM interrupt hooks so that programs written to use CPM could be 'ported' to MS-DOS, but Microsoft quickly deprecated the practice and provided less and less support for this executable file format.

Nevertheless, unless you have an 8080/8085 cross-assembler that runs under Windows, XASM85 is the only means to produce the necessary object file.

To run it today involves using an MS-DOS emulation program called DOSBox, and a search of the Internet shows that it can be obtained from:

www.dosbox.com

If you wish to use this emulator with other MS-DOS legacy software, for example GW-BASIC, then you may wish to install the optional shell. I use DOSShell, and the link for it is also available on the same website.

Even with DOSBox, XASM85 is somewhat quirky. If you run it from the command line, then do not specify a listing file. For some reason specifying a listing file this way seems to affect its operation, and it may simply terminate without producing anything.

If you specify the files from the command line, then this seems to work better, for example:

```
XASM85 DEBUG.ASM DEBUG.LST DEBUG.HEX
```

if you do not require the listing file, then specify the null device thus:

```
XASM85 DEBUG.ASM NULL: DEBUG.HEX
```

XASM85 will produce an ASCII HEX object file. The first line contains the start, end, and execution addresses in decimal, separated by spaces. Then follows all the object file in pairs of ASCII HEX digits, with 80 upper-case characters per line. The final object code line may well be shorter.

The last line of the file is the decimal checksum. This is formed by summing the start, end, and entry point addresses along with the actual decimal value represented by each byte of the load file.

In order to use the file with DBMAK.DO, edit the file by putting commas between the start, end, and execution addresses, and removing the surplus spaces.

Note that the file is not Intel HEX!

To make the CO file, rename the hex object file with the .DO extension, and download it to the Tandy 100. Download the DBMAK.DO file in the same way. You will need to ensure that there is sufficient memory available, and it would be wise to start with a clear RAM.

Enter BASIC and enter the command:

```
CLEAR 512,xxxxxx
```

where xxxxx is replaced by the start address in decimal.

The run the program. It will prompt you for the source file name, and then it will read the file and convert each hex digit pair into a byte of machine code and poke it into the appropriate address in RAM. You will be prompted at the end if you wish to save the file.

At this point you can delete the BASIC program and the hex load file. From the MENU place the cursor over the DEBUG.CO file, and press the enter key, and you should see the DEBUG sign-on screen.

An alternative is to make the DEBUG.DO re-constructor program. To do this, you will need to use the HEXDAT.BAS GW-BASIC program to convert the hex object file into data statements. Run GW-BASIC and load HEXDAT.BAS. Run the program and follow the prompts, and the result will be the ASCII text file containing the DATA statements.

Load DEBUG.DO into your text editor, remove any existing data statements and insert the new data statements just constructed. Edit the top line to your preference, and save the result.

Download this file to the Tandy Model 100. You will need to clear all of RAM to accommodate this file. Load it into BASIC, and then kill the file. Run the file and it will re-construct DEBUG.CO for you without the necessity of you having to calculate offset addresses, nor having to set the high memory address. It will take about 6 minutes for this to execute.

It is not the purpose of this document to try and teach you how to use DEBUG, you can read the User Manual and 'play' with it to familiarise yourself with its operations.

The next phase concerns what steps you need to take if you decide to modify the code for your own purposes.

Obviously you will need to modify the assembly language source file, and re-assemble it. You can then follow the previous steps to reconstruct the machine language image file in the Model 100. However without the re-locator program you would have to re-assemble and reload the program every time you wished to move it to a different location so that it could co-exist with other terminate and stay resident machine language software such as TS-DOS, for example.

It was for this reason that I decided to design and write DBCFG.DO. This is the BASIC program to patch the image in memory so that it can run at an address different from its original origin address.

To re-locate the code, load DBCFG.DO in the same way, enter BASIC, load the program, and run it.

It will search the directory table to find the DEBUG.CO file's location, and if it exists, it will then check to see if it is the correct version of DEBUG.

Since you will be patching machine code addresses, it is obvious that the file must exactly match that expected by the program.

If the file exists and matches, then you will be prompted for the new location which must be higher than address 32512 and no higher than the original start address. Assuming the new location is OK, you will be prompted whether you wish to continue, and if you answer Y or y, then the patching will commence.

At the conclusion your .CO file will have been patched to run at its new address, and you will have to enter BASIC and use the CLEAR command to set the new high memory pointer.

The re-location operation is fully reversible, you can re-locate the program any number of times.

If you examine DBCFG you will note that a major portion is occupied with BASIC DATA statements. These statements contain the offset address from the start of the DEBUG.CO file in memory that refer to the various local addresses used for jump and call instructions. To illustrate, this is the assembly language at the start of DEBUG:

```
entrpt      lxi    h,entrpt-020h
            sphl                    ;Set stack pointer
            shld   pssp              ;Update pseudo-sp
            call   clrscr            ;Clear the screen,
            lxi    h,cpyrt          ;and blow my own trumpet.
            call   prtstr
```

The symbol `entrpt` is obviously local to DEBUG, as are the symbols `pssp` and `cpyrt`. The symbols `clrscr` and `prtstr` are ROM locations, and obviously do not change.

Thus, when DEBUG is assembled, the local symbolic addresses will change depending upon the value assigned to the origin statement for the code; the non-local ROM symbolic addresses on the other hand will remain the same.

The ‘trick’ to making the re-locator DATA table work is to assemble one version of DEBUG at its normal address in high memory, and a second version exactly 256 bytes lower. Examples are the DEBUGH.HEX and DEBUGL.HEX files.

Why ‘exactly 256 bytes lower’ you might well ask, and the reason is that this will generate an address for local symbols whose most significant byte differs from the ‘normal’ version of DEBUG by exactly 1, and whose least significant byte remains the same. In this way, you only have to search through the two HEX load files for single bytes that differ.

In 8085 object code, the address references are stored in low-byte/high-byte order, and thus the upper byte is the last byte of a jump or call instruction. For the re-locator program however we need to modify the entire word, and thus we need the address or offset of the previous byte.

Again, to give an example, note that the first instruction loads the HL register pair with the address of the stack pointer, and this address is 32 bytes (20h) lower than the entry point. Thus, since DEBUG version 4.13 is origin’ed at address E2D9h, which is also the entry point, the actual code for this line is:

```
entrpt      lxi    h,e2b9h
```

If you disassemble DEBUG, you will see this code. Suppose we origin the code at exactly 256 bytes lower, then disassembling this new version of DEBUG reveals:

```
entrpt      lxi    h,e1b9h
```

The high-order byte has changed by 1.

You may well ask, ‘Why not simply change the origin address by 1 instead?’ In this case all the low order bytes will certainly change by 1, but what happens if a local address reference were to lie exactly on a 256 byte boundary?

Suppose a call or jump instruction referenced address D000 for example. If you now re-located the program down by one byte, this address would change to CFFF changing both the lower and upper bytes, making it much more difficult to extract a difference file from the HEX load file.

Having now re-assembled DEBUG to reside exactly 256 bytes lower in memory, you can use the MKDATA.BAS program using GW-BASIC or QuickBasic to generate the line numbered DATA statements that will be incorporated into the DBCFG program.

Unfortunately there is not enough memory in a Model 100 to permit you to do this on the target machine and write out the DATA file as well.

As a note, GW-BASIC is sufficiently well-behaved that it will indeed run under a Windows CMD window, as will QuickBasic, so the DOSBox program is not essential for these operations. Nevertheless, it is still useful to use it to ensure complete compatibility.

To generate the DATA statements, first prepare the HEX load files. This involves removing the first line with the start, end, and entry point addresses. Then remove the last line containing the checksum, and save the files.

Load the program MKDATA into whichever BASIC you have, and run it. Respond to the prompts, and the program will very quickly generate the DATA statements.

Since I haven’t bothered to use any fancy formatting, the DATA statement file could well do with being ‘trimmed’ with a text editor. I use Notepad++ and using this editor, I have a macro that imports the DATA.TXT file, and removes the leading and trailing spaces, and replaces the space-comma-space between the data values with a single comma.

Note, this ‘trimming’ is not essential, the BASIC interpreter in the Model 100 will ignore the extra spaces anyway, and, after you have loaded DBCFG.DO into the interpreter you can simply delete the DBCFG.DO file, the tokenised BASIC program will still reside in the BASIC interpreter’s workspace until over-written with another program, or a NEW command is executed.

The tricky part is now patching DBCFG.DO. Load it into your favourite editor, and remove the existing data statements. Open the DATA.TXT file, and copy the entire DATA statement code into the end of the DBCFG.DO code.

Now comes the tricky bit. There are three data constants that need changing. The first is the signature bytes you intend to use. I used the ASCII version number. The second is the offset from the start of the file to where these signature bytes are located, and the third is the start address of your new version, that is, the high memory version you assembled first. The important line in the DBCFG file is:

```
10 CLEAR512,MAXRAM:VN$="4.13":VO=4503:OA=58073
```

In this case, the 'signature' is the ASCII version: "4.13" you will need to change this to your version string. The variable VO is the Version Offset address.

To find this, load and execute DEBUG.CO and use the search command to locate the address of your version string. Make a note of the address. Then use the C command to calculate the difference between the start address and the offset address. Convert this to a decimal value, add 6, and insert in the DBCFG file.

The addition of 6 is necessary because the re-configuration program has to calculate the offset from the beginning of the file, rather than from the address in memory, and the file has 6 bytes pre-pended corresponding to the top, end, and entry point addresses.

Finally, alter the variable OA (Origin Address) to correspond to your decimal origin address.

If everything has gone according to plan, then you should now have a working version of DBCFG.DO which will be able to re-locate your new version of DEBUG to wherever you wish in memory.

Of course, there is no reason why you could not insert a suitable binary signature word at the start of the file thus obviating the need to perform a search and calculation exercise. All that would be necessary then would be to recover the decimal value of this signature word in the usual way, and also set the entry address appropriately.