DEFT Debugger

Debug

1 Introduction

The **DEFT Debugger** is a software module which can be linked into any program produced by **DEFT** software products. It becomes the *main* module in the resulting program and allows the programmer to control its resulting execution. **DEFT Debugger** includes the following features:

- Like other debuggers, this one provides for memory and register display and modification as well as instruction breakpoints. Memory display and modification can occur in hex, decimal, floating point, ASCII and string formats.
- Single Pascal statement execution is available when the DEBUG option is specified at compile time.
- Normal program operation can be interrupted and the Debugger activated when the *BREAK* key is depressed.
- Symbolic access to memory areas is automatically provided by a special interface to the **DEFT Pascal** Compiler. This symbolic access includes automatic as well as static variables.
- A general expression capability allows the Debugger to perform all arithmetic and type and base conversions for you.
- A trace facility provides you with a procedure call history so that you can see how you got to a specific point in a Pascal program.
- Automatic screen preservation restores the screen area and attributes anytime program execution is resumed. This simplifies debugging of graphic programs.

2 General Operation

Although there are a number of features built into the **DEFT Debugger** specifically to debug Pascal programs, any program produced with DEFT software products can be debugged with it.

2.1 Linking in DEFT Debugger

In order to use **DEFT Debugger** you answer the **DEFT Linker's** *DEBUG? (Y)* question with anything other than *N* or *n* when you link the program. **DEFT Debugger** is automatically included in the resulting binary and gets initial control of the 6809 microprocessor when your program is executed. **DEFT Linker** provides **DEFT Debugger** with a table of all the module names and offsets in the resulting program along with the address where your program would normally begin execution. **DEFT Debugger** is loaded, as a part of your program, when you load your program with the *LOADM "myprogm":EXEC* command.

2.2 Debug Screen

After linking your program you are ready to execute it. When you begin execution **DEFT Debugger** will gain control and present you with its screen. This initial screen looks like this:


```
SYMBOLIC ONLINE DEBUGGER V3.x
(C) 1983 DEFT SYSTEMS, INC.
COMMAND:
 PS  02     B0 0000
 VD  00     B1 0000
 VC  00     B2 0000
            B3 0000
 CC  xx     B4 0000
 A   xx     B5 0000
 B   xx     B6 0000
 DP  xx     B7 0000
 X   xxxx
 Y   xxxx
 U   xxxx
 PC  xxxx
 S   xxxx
```

DEFT Debugger is now waiting for a command to execute and has displayed the complete set of registers it maintains for the program being debugged. You will normally enter a two character command. **DEFT Debugger** then prompts you for any additional

parameters required by the particular command.

The chapter on *Commands* describes all the commands and their required parameters. The chapter on *Expressions* describes the rules for forming expressions which are used in most parameters. What you see on the screen when the Debugger is first activated or anytime you hit a breakpoint is the automatic execution of the *DR* command. Following is a short description of the types of operations for which you might use **DEFT Debugger**.

2.3 Setting Breakpoints

One of the first things that you will want to do with the Debugger will be to set a breakpoint. A breakpoint is a place in your program where you want your program's execution to be suspended and **DEFT Debugger** activated. This allows you to examine variables or in the case of assembler language, registers. You can then see if the program has produced the proper intermediate results.

You set a breakpoint by using the Debugger's *modify register* command to set the value of one of the eight Breakpoint registers to the address of the place in your program where you want the breakpoint to occur. You have 8 breakpoint registers which allows you to specify up to 8 different places in your program at one time. This is especially convenient when you are not sure which place your program will go to first. The section on *Symbols* under *Expressions* describes how to specify a symbolic address.

2.4 Executing Your Program

After having set some (possibly no) breakpoints, you may then use **DEFT Debugger**'s *GO* command to begin (or continue) your program's execution. Another possible command is **DEFT Debugger**'s *ST* (Single Step) command which will allow you to specify the number of Pascal statements that you want to execute. Note that this option is only available when you have previously enabled the *debug?* option when the Pascal program was compiled.

2.5 Interrupting Program Execution

If you used the *GO* command to start execution, it will stop executing when one of the breakpoints that you specified is encountered. If you used the *ST* command, then execution will stop when the specified number of Pascal statements have been executed.

In either case you may stop the program's execution by depressing the *BREAK* key. If the program was compiled with the *DEBUG* option enabled then execution will stop on the next Pascal statement that is executed. Depressing the *BREAK* key while the program is prompting for keyboard input will cause it to stop even if the Debug option was not enabled at compile time.

2.6 Displaying/Modifying Memory and Registers

After your program stops, the Debugger is re-activated and you can use the display commands to determine what your program has done so far. You can change any variable or register that you wish before resuming execution again in order to change the way that your program is executing. Note that if your program stopped because it encountered a breakpoint that you specified via one of the breakpoint registers, then you will have to clear that breakpoint before resuming your program. Otherwise, the program will immediately breakpoint again.

2.7 Checking Program State

In addition to variables (memory) and registers, you can also use the *US* (User Screen) command to see what the screen is supposed to look like when **DEFT Debugger** is not using it. In addition, the *TR* (TRace) command will follow the chain of pointers that Pascal builds on the stack. This trace of all the current activation blocks will tell you what Pascal procedures are currently active and where they were called from.

3 Commands

This section describes all the commands available on **DEFT Debugger**. The title of each subsection names the corresponding command and contains the two character command representation in parentheses.

3.1 Display Register (DR)

This command causes all the **DEFT Debugger** registers to be displayed. All registers are displayed in hexadecimal. Those which are 16 bit registers are also displayed as module offsets with the module name and hex offset displayed following the absolute hex value.

The registers B0 through B7 are the breakpoint registers which can be set to addresses in your program at which you want execution to stop. The registers CC, A, B, DP, X, Y, U, PC and S are the 6809 machine registers. The remaining three registers relate to the graphic capabilities of the TRS-80 Color Computer and are as follows:

- *PS* is the Page Select register. The *lower* 7 bits of this register specify the *upper* 7 bits of the memory address at which the screen page begins. This value is initially 2 indicating that the screen page begins at address $400 or 1024.
- *VD* is the Video Display Generator register. The lower 3 bits of this register specify the graphics mode that is to be used.
- *VC* is the Video Control register. The *upper* 5 bits of this register specify the color set and qualify the graphics mode selected by the VDR.

Unlike the 6809 registers, the graphics registers cannot be read and saved by **DEFT Debugger**. Therefore anytime your program modifies these values at a point at which you are breakpointing, you will have to tell the Debugger what these values should be. This is done via the *Modify Register (MR)* command.

3.2 Display Word (DW)

This command allows you to display 1 or more 16 bit words in memory in both decimal and ASCII formats. There are two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the first 16 bit word to display.

- *COUNT:* - This parameter requires an expression which specifies the number of 16 bit words to display. If you enter nothing then the count defaults to 1.

3.3 Display Byte (DB)

This command allows you to display 1 or more 8 bit bytes in memory in both decimal and ASCII formats. There are two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the first 8 bit byte to display.
- *COUNT:* - This parameter requires an expression which specifies the number of 8 bit bytes to display. If you enter nothing then the count defaults to 1.

3.4 Display Floating Point (DF)

This command allows you to display a Pascal floating point (real type) number variable. There is one parameter:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the floating point variable.

The floating point variable is displayed in decimal format.

3.5 Display String (DS)

This command allows you to display a Pascal string variable. There is one parameter:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the string variable.

The string variable is displayed in ASCII format. In addition, the decimal length of the string is displayed.

3.6 Display Variable(DV)

This command allows you to display a variable as either a word, byte, floating point or string. You must use a symbol as part of the *ADDRESS* parameter. **DEFT Debugger** uses the type of the symbol used to determine which type of display to perform. There are two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the variable.
- *COUNT:* - This parameter is prompted for only when the symbol type is an ARRAY. It requires an expression which specifies the number of 8 bit bytes or 16 bit words to display. If you enter nothing then the count defaults to 1.

3.7 Display Hex (DH)

This command allows you to display 80 bytes of memory in both hex and ASCII representation. There is one parameter:

- *ADDRESS:* - This parameter requires an expression which specifies the address of memory to begin the display.

This command displays the memory as 10 lines of 8 bytes each. The last 3 hex digits of the memory address is displayed at the beginning of each line followed by the hex representation of the 8 memory bytes at that location. Finally, the ASCII representation of those same bytes is displayed at the end of the line.

3.8 Display Next (DN)

This command is almost exactly the same as Display Hex (DH) except that you are not prompted for an address. The display begins at the point where the last Display Hex or Display Next left off. This command provides a convenient means to page through memory.

3.9 Modify Register (MR)

This command allows you to modify any of **DEFT Debugger**'s registers. All registers displayed on the Display Register screen can be modified. This command has two parameters:

- *REGISTER:* - This parameter requires the 1 or 2 character name of the register that is to be modified.
- *VALUE:* - This parameter requires an expression which is the value that the register is to be set to.

3.10 Modify Word (MW)

This command allows you to modify a 16 bit word in memory. It requires two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the 16 bit word to modify.
- *WORD xxxx VALUE:* - This prompt shows the hexadecimal address that will be modified (the "xxxx"). It requires an expression which specifies the value that the word at that location is to be set to. If nothing is entered, the command is terminated and the word is not modified. If a value is entered, then the word is modified and **DEFT Debugger** continues to prompt for subsequent words until nothing is entered.

3.11 Modify Byte (MB)

This command allows you to modify an 8 bit byte in memory. It requires two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the 8 bit byte to modify.
- *BYTE xxxx VALUE:* - This prompt shows the hexadecimal address that will be modified (the "xxxx"). It requires an expression which specifies the value that the byte at that location is to be set to. If nothing is entered, the command is terminated and the byte is not modified. If a value is entered, then the byte is modified and **DEFT Debugger** continues to prompt for subsequent bytes until nothing is entered.

3.12 Modify Floating Point (MF)

This command allows you to modify a Pascal floating point (real type) number variable in memory. It requires two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the floating point number to modify.
- *VALUE:* - This parameter requires the decimal representation of the floating point value that is to be inserted.

3.13 Modify String (MS)

This command allows you to modify a Pascal string in memory. It requires two parameters:

- *ADDRESS:* - This parameter requires an expression which specifies the address of the string to modify.
- *xxxx STRING:* - This parameter requires a number of ASCII characters to be entered. These are stored directly in the string with the number of characters entered becoming the string's length. If nothing is entered, the command is terminated and the string is not modified.

3.14 Modify Variable (MV)

This command allows you to modify a Pascal variable by identifying it symbolically. This command allows **DEFT Debugger** to determine whether to execute a *Modify Word, Modify Byte, Modify Floating* or *Modify String* command depending on the type of the variable named in the *ADDRESSS:* parameter.

3.15 Clear Breakpoints (CB)

This command is used to clear all the breakpoint registers to zero. You can set a breakpoint by using the Modify Register (MR) command to set one or more of these registers to a non-zero value. You can also clear an individual breakpoint by using the same command to set a breakpoint register to zero.

3.16 User Screen (US)

This command allows you to view the screen currently being displayed by the program under test. The values of the PS, VD and VC registers are used to determine what the display is to look like. The display persists until you type any character.

3.17 Evaluate (EV)

This command allows you to evaluate an expression and display its results in decimal, hexadecimal and ASCII. It requires one parameter:

- *VALUE:* - This parameter requires an expression which is to be evaluated.

3.18 Trace (TR)

This command allows you to see all the procedures which are currently active. The absolute address and module offset of the current program counter (PC) and each return address on the stack (beginning with the most recent) is displayed on each line. For those modules which also have symbols, the name of the procedure or function to which the return address points is also displayed. This then provides you with a list of each active Procedure/Function and the point in the calling Procedure/Function from which they were called.

Since this command relies on the standard Pascal frame structure, there are some limitations on its use:

- Only those Procedure/Function activations that have been completed will be displayed. If you set a breakpoint at the address of a Procedure or Function and then do a *TR*, you will not see that Procedure or Function in the list. You must set the breakpoint at (or Single Step to) the first statement in the Procedure or Function. Note that the Single Step *(ST)* command will not breakpoint in the middle of a Procedure/Function activation (unless you have set an explicit breakpoint).
- The command is not meaningful until after the complete activation of the main Pascal program. This is done the same as a Procedure or Function described above.
- Only the most recent 12 (or fewer) activations are listed.
- Calls to Assembly language routines will be listed only if they construct a Pascal frame structure on the stack.

3.19 Go (GO)

This command allows you to execute your program. If any of the breakpoint registers are non-zero then breakpoints are set at those points before program execution begins. **DEFT Debugger** will not regain control until one of the specified breakpoints is encountered. If one of the breakpoints is the same as the PC register then control will return immediately to the Debugger. This command has no parameters.

Once a breakpoint is encountered, the DR command is automatically executed and you are prompted for another

command.

3.20 Step (ST)

This command is similar to the *GO* command except that it uses the breakpoints inserted into the program by Pascal when you specified *debug* at compile time. Not only does the **DEFT Pascal** compiler include symbol tables, but it also generates a breakpoint instruction at the beginning of every Pascal statement when you specify the *debug* option. The *Step* command then lets you step through the Pascal statements by counting the corresponding breakpoints in the resulting code.

Note that this command will operate the same as the GO command if there are no Pascal modules with the *debug* option enabled. This command has 1 parameter:

- *COUNT*: - This parameter requires an expression which is the number of Pascal statements to execute before returning control to **DEFT Debugger**. If no expression is entered, a value of 1 is assumed.

3.21 Quit (QU)

This command allows you to terminate your program and return control to BASIC.

4 Expressions

Most DEFT **Debugger** commands will prompt you for some additional information such as an address of a field or a value which is to be used by the command. Most of these additional prompts require a general expression to be entered. This expression can be as simple as a single digit or as complex as several numbers in various bases with symbols combined with different operators. This section describes the rules for forming these expressions.

The **DEFT Debugger** deals entirely in 16 bit units. All components of an expression have 16 bit values and any resulting expression also has a full 16 bit value.

4.1 Constants

A constant used by itself is a legal expression. The **DEFT Debugger** supports 4 types of constants.

1. A *decimal* constant is a set of numbers in the range of -32768 to 32767.
2. A *hexadecimal* constant is a dollar sign ($) followed by up to 4 hexadecimal digits (0..9,A..F). If the constant is less than 4 digits long, leading zeroes are assumed.
3. An *ASCII* constant is a single quote (') followed by a single ASCII character. The value of this constant is the binary value of the ASCII character as the low 8 bits with the high 8 bits being zero.
4. A *double ASCII* constant is a double quote (") followed by two ASCII characters. The value of the constant is the binary value of the first ASCII character as the high 8 bits and the second as the low 8 bits.

4.2 Registers

The current contents of any of the registers can be referenced by entering a percent sign (%) followed by a one or two character register name. The available registers are those displayed via the Display Register (DR) command. They are as follows:

Mnemonic	Bit Size	Description
PS	8	Page Select
VD	8	Video Display Generator
VC	8	Video Control
CC	8	6809 Condition Code
A	8	6809 Accumulator A
B	8	6809 Accumulator B
DP	8	6809 Direct Page
X	16	6809 Index X
Y	16	6809 Index Y
U	16	6809 User Stack
PC	16	6809 Program Counter
S	16	6809 System Stack
B0	16	Breakpoint 0
.		
.		
.		
B7	16	Breakpoint 7

4.3 Symbols

Symbols are the names or *identifiers* that you used in your source code program to reference variables, procedures and functions. If the program that you are debugging has some Pascal modules in it, you can have the compiler include the symbols found in these modules by answering its *DEBUG?* prompt with anything other than *N* or *n*. This will cause the compiler to include the names of all the variables, procedures and functions in specially formatted tables. These tables are imbedded in the resulting object module code.

Object modules created with this option will be larger due to the presence of the symbols which will be part of the final load module binary code. When you have several Pascal modules in a single program, you can reduce the symbol table memory requirements by specifying debug symbols in only the modules that you wish to debug. The debugger knows which modules have symbols and which ones don't so that you only get the symbols that you need.

There are three types of symbols which are referenced in three different ways:

1. A *Module* symbol is the filename (not including the extension) of an object file or library section which was linked with the debugger. You indicate a module symbol with a leading less than sign (<) followed by the symbol itself. The names of all the object modules that are linked together are known to **DEFT Debugger** regardless of whether symbols internal to the corresponding module are present. This means that you can use module symbols even with assembly language modules. The value of a module symbol is the absolute memory address of the first instruction at the beginning of the module.

   One of the most common uses of a module symbol is to specify an address within a module. This is usually done as follows:

   <MYMODULE+$1A3

   This form can be used to set breakpoints in either Pascal or assembly language modules. In this case 01A3 is the offset within the module where the Pascal statement starts on which you want to breakpoint.

2. A module symbol can be further qualified with a *static symbol*. This is done by immediately following the module symbol with a greater than sign (>) followed by the static symbol. This static symbol can represent any Pascal procedure, function or statically allocated variable. The value of a static symbol is the beginning memory address of the program element represented by the symbol.

   A static symbol can be further qualified to any level required by entering additional greater than signs (>) followed by the qualifier. For example:

   <MYMODULE>UTILPROC>LCLFUNC>X

   This entry specifies the static symbol *X*, which is local to the function *LCLFUNC* which is contained within the procedure *UTILPROC*. This procedure in turn is in the module *MYMODULE*.

   After a module has been referenced (either by itself or as part of a static symbol reference) the next static symbol can be specified without specifying that same module name. **DEFT Debugger** will use the last module referenced, as a basis for its search, anytime a static symbol is specified without a leading module name.

3. An *automatic* symbol is indicated when a leading alphabetic character is detected. In this case **DEFT Debugger** will automatically scope the symbol by following the static procedure call links in the stack. This type of symbol specification will find the symbol which is known at the current point in the program. You can use this type of specification for procedure, function and static variable symbols as well as automatic variable symbols.

4.4 Terms and Indirection

The elements or arguments of an expression, *constants*, *registers* and *symbols*, are generically known as *terms*. You can add a level of *indirection* to a term by prefixing it with an at sign (@). This means that the *value* of the term is used identify the location of, or to *address*, a 16 bit word in memory. The contents of that memory word are then used as the value of the term. This is known as an *Indirect Term*.

4.5 Operators

Terms and Indirect Terms can be combined with the use of *operators*. The operators which are available are the four arithmetic operators: addition (+), subtraction (-), multiplication (*), and division (/). There is no precedence between operators and all expressions are evaluated from left to right.


Debug