MICROWARE®

INTERACTIVE DEBUGGER

USER'S MANUAL

Microware Systems Corporation
5835 Grand Avenue
Des Moines, Iowa 50312
Telephone 515-279-8844
Telex 910-520-2535

INTERACTIVE DEBUGGER

USER'S MANUAL

Fourth Revision
Publication Date: June 12, 1982

Microware System Corporation
5835 Grand Avenue
Des Moines, Iowa 50312 U.S.A.
Telephone 515-279-8844
Software Support 515-279-8898
Telex 910-520-2535

MEMORY COMMANDS

The interactive debugger has two memory-related commands to display large blocks of memory, and to clear and test memory.

Clear and Test Memory Command

The command C followed by TWO expressions simultaneously performs a "walking bit" memory test and clears all memory between the two evaluated addresses. The first expression gives the starting address, and the second the ending address (which must be higher). If any byte(s) fail the test, its address is displayed. Of course, only RAM memory can be tested and cleared.

WARNING: THIS COMMAND CAN BE DANGEROUS FOR OBVIOUS REASONS. BE SURE OF WHAT MEMORY YOU ARE CLEARING.

Example:

```
        DB: C 1200 15FF
        DB: C . .+256
```

Dump Memory Command

The M command, which is also followed by two addresses, displays a screen-sized display of memory contents in tabular form, in both hexadecimal and ASCII form. The starting address of each line is printed on the left, followed by the contents of the 16 subsequent memory locations . On the far right is the ASCII representation of the same memory locations. Those locations containing non-displayable characters have periods in their place. The high order bit is ignored for the display of the ascii character.

Search Memory Command

The "S" command is used to search an area of memory for a one- or two-byte pattern. The search begins at the present Dot address. The "S" is followed by two expressions: the first expression is the ending address of the search, and the second expression is the data to be searched for. If this value is less than 256 a one-byte comparison is used, otherwise two bytes are compared. If a matching pattern is found in memory, Dot is set to the address where it was located (which is displayed). If no match occurred, another "DB:" prompt is displayed.

This Page Intentionally Left Blank

OS-9 INTERACTIVE DEBUGGER USERS MANUAL
Table of Contents

This Page Intentionally Left Blank

INTRODUCTION

The Microware Interactive Debugger is a powerful tool for system diagnostics or testing 6809 machine language programs. It is also useful when you need to directly access the computer's memory for any of a number of reasons: testing I/O interfaces, verifying data, etc. The calculator mode can simplify computation of addresses, radix conversion, and other mathematical problems often encountered by machine-language programmers.

Basic Concepts

The debugger operates in response to single line commands typed in from the keyboard. You can tell when you are "talking to" the debugger because it always displays a "DB:" prompt when it expects a command line.

Each line is terminated by a carriage return ("new line" on some keyboards). If you make a mistake while typing, you can use the backspace key (control H), or delete the entire line using the control-X key.

Each command line starts with a single character command which may be followed by text, or one or two arithmetic expressions, depending on the specific command. Upper-case and lower-case character can be used interchangeably. Here's an example of the "space" command which displays the result of an expression in hexadecimal and decimal notation:

```
DB:  A+2
    $000C  #00012
DB:
```

Numbers entered into or displayed by the debugger are in hexadecimal notation, unless special commands are used, such as the decimal conversion command shown above. Two important topics must be covered before beginning the command descriptions themselves; expression syntax and DOT.

This Page Intentionally Left Blank.

EXPRESSIONS

A powerful capability of the Interactive Debugger is its integral expression interpreter, which permits you to type in full expressions wherever an input number is called for in a command. Expressions used by the Interactive Debugger are similar to those used with high-level languages such as BASIC, except there are some extra operators and operands that are unique to the debugger.

Numbers used in expressions are 16 bit unsigned integers, which is the 6809's "native" arithmetic representation. The allowable range of numbers is therefore zero to 65535. Two's complement addition and subtraction is performed correctly, but will print out as large positive numbers in decimal representations. Some commands require byte values and an error message will be given if the result of the expression is too large to be stored in a byte (> 255). Also, some operands are only a byte long (such as individual memory locations and some registers). These are automatically converted to 16-bit "words" without sign extension. Spaces may be used between operators and operands as desired to improve readability but do not affect evaluation.

Constants

Constants can be in base 2 ("binary"), base 10 ("decimal"), or base 16 (hexadecimal or "hex"). Binary and decimal constants require a prefix character: % (binary) or #(decimal). All other numbers are assumed to be hex. Hex numbers may also have an optional $ prefix. Here are some examples:

DECIMAL	HEXADECIMAL		BINARY
#100	64	$64	%1100100
#255	FF	$FF	%11111111
#6000	1770	$1770	%1011101110000
#65535	FFFF	$FFFF	%1111111111111111

Character constants may also be used. A single quote ' for one character constants and a double quote " for two-character constants. These produce the numerical value of the ASCII codes for the character(s) which follow. For example:

```
'A   = $0041
'0   = $0030
"AB  = $4142
"99  = $3939
```

Special Names

There are other legal operands in expressions; Dot, Dot-Dot, and register names.

Dot is simply the debugger's current working address in memory. It can be examined, changed, updated, used in expressions or recalled. It has the main effect of eliminating a tremendous amount of memory address typing. The following debug command prints the current working address:

```
DB: .
    2204 82
```

Dot-Dot is the previous value of Dot, if it was changed. The following debug command prints the previous value of Dot:

```
DB: ..
    0500 12
```

Register Names

MPU Registers may be specified by a colon character ":" followed by the mnemonic name of the register; for example:

:A	Accumulator A
:B	Accumumator B
:D	Double Accumulator
:X	X Register
:Y	Y Register
:U	U Register
:DP	Direct Page Register
:SP	Stack Pointer
:PC	Program Counter
:CC	Condition Codes Register

The values returned are of the program under test's registers, which are "stacked" when the debugger is active. Those registers which are a single byte long are promoted to a word when used in expressions.

NOTE: When a program is interrupted by a break point, the SP will be pointing at the bottom of the MPU register stack.

Operators

Operators indicate arithmetic or logical operations. The operators having a higher precedence are executed before those having lower precedence. For example, all multiplications are performed before additions. Operators in a single expression having equal precedence are evaluated left to right. Parentheses may be used to override precedence as desired. Here are the operators, in precedence order from weaker to stronger:

+	addition	-	subtraction
*	muliplication	/	division
&	logical AND	!	logical OR
-	nega		

This Page Intentionally Left Blank

DEBUG COMMANDS

Calculator Command

To use the calculator command, enter a line starting with one or more spaces followed by any legal expression, then "return". The expression is evaluated and the result is displayed on the following line in both hexadecimal and decimal representations.

Here are some examples:

```
DB:  5000+200
    $5200 #20992

DB:  8800/2
    $4400 #17408

DB:  #100+#12
    $0070 #00112
```

These commands are also handy for converting values from one representation to another:

```
DB:  %11110000
    $00F0 #00240
DB:  'A
    $0041 #00065
DB:  #100
    $00C4 #00100
```

You can also use indirect addressing to look at memory without changing Dot:

```
DB:  <.>
    $004F #00079
```

Another trick is simulating 6809 indexed or indexed indirect instructions. For example:

```
DB:  [:D+:Y]
```

is the same as the assembly language syntax [D,Y].

Memory Examine

Several commands relate directly to Dot. For example, typing just "." causes the current value of Dot and the contents of that memory address to be displayed, for example:

```
      DB: .
          2201 B0
      DB:
```

The first number, 2201, is the present value of Dot, and B0 is the contents of memory location 2201. Typing a line with nothing (just a return) increments Dot and prints its new value and contents. This is how to "step through" sequential memory locations:

```
      DB:
          2202 05
      DB:
          2203 C2
      DB:
          2204 82
```

The minus sign is the opposite: it decrements Dot and prints its value and contents:

```
      DB: .
          2204 82
      DB: -
          2203 C2
      DB: -
          2202 05
```

To change the value of Dot, you type a period, followed by an expression, which is evaluated and becomes the new value for Dot:

```
      DB: . 500
          0500 12
```

Memory Change

To change the contents of the memory location pointed to by Dot, you type an equal sign followed by an expression. The expression is evaluated, and the result stored at Dot, which is then incremented and the next address/contents are displayed. The memory location is checked after the new value is stored to make sure it actually changed to the correct value. If it didn't, an error message is displayed. This will happen when an attempt is made to alter non-RAM memory. In particular, many 6800-family interface devices (such as PIAs, ACIAs, etc.) have registers that will not read the same as when written to.

```
DB: .
    2203 C2
DB: =FF
    2204 01
DB: -
    2203 FF
DB:
```

One additional feature: whenever Dot is changed, its last value is saved, and can be restored by typing two periods:

```
DB: .
    1000 23
DB: . 2000
    2000 9C
DB: ..
    1000 23
```

REGISTER COMMANDS

Several forms of the register command can be used to examine one or all registers, or to change a specific register's contents. A word about registers: the registers accessed are "images" of register values on a stack. When the debugger is first entered, an initial stack is automatically set up for the user. The register images on this stack are passed to the program under test the first time the "G" command is used. The registers are also valid after breakpoints are encountered and are passed back to the program upon the next "G" command.

IMPORTANT NOTE ABOUT CHANGING REGISTER CONTENTS:

1. IF YOU CHANGE THE SP REGISTER, YOU WILL MOVE YOUR STACK AND THE OTHER REGISTER CONTENTS WILL CHANGE.

2. BIT 7 OF THE CC REGISTER (THE E FLAG) MUST ALWAYS BE SET FOR THE G COMMAND TO WORK. IF YOU FORGET, THE DEBUGGER WILL NOT RETURN TO THE PROGRAM CORRECTLY.

Displaying Registers

To display the contents of a specific register, enter a colon ":" followed by the register name. The Debugger will respond by displaying the current register contents in hex. Examples:

```
DB: :PC
    C499
DB: :B
    007E
DB: :SP
    42FD
```

To display all registers, type ":", then hit return. The debugger responds by displaying the register names with their corresponding hex contents beneath.

```
DB: :
      SP   CC  A  B  DP   X     Y    U    PC
    C499  C4  20 1C  01 DD3E 239A 0000 240C
```

Changing Register Contents

To assign a new value to a register, type the register name followed by an expression. The expression is evaluated and stored in the register specified. When 8-bit registers are named, the expression given must have a value that fits in a single byte, or an error message is displayed and the register is not changed.

Here are some examples:

```
DB: :X #4096

DB: :DP 0

DB: :D 24CF+:Y
```

Breakpoint Commands

The breakpoint capabilities of the debugger allow you to specify addresses where execution of the program under test is to be suspended, and the debugger re-entered. When a breakpoint is encountered, the values of the MPU registers and the "DB:" prompt will be displayed meaning the debugger is ready to accept a command. Registers can be examined or changed, memory can be altered, etc. Breakpoints may be inserted at up to 12 different addresses.

Breakpoints are implemented by using the 6809 SWI instruction, which when executed, interrupts the program and saves its complete state on the stack. The SWI instructions are automatically inserted and removed by the debugger at the right times so you will not "see" them in memory. Because the SWIs operate by temporarily replacing an instruction opcode, there are three restrictions on their use:

1. Breakpoints cannot be used in programs in ROM.

2. Breakpoints must be located in the first (opcode) byte of the instruction.

3. User programs cannot utilize the SWI instruction for other purposes (but CAN use SWI2 and SWI3)

When the breakpoint is encountered during execution of the program under test, the debugger is reentered and the program's register contents are displayed using the same format as the "display register" command.

Setting Breakpoints
Display Breakpoints

The B command is used to insert a breakpoint if followed by an expression, or to display all present breakpoint addresses if given without an expression.

```
DB: B 1C00
DB: B 4FD3
DB: .
    1277 39

DB: B .
DB: B
    1C00 4FD3 1277
DB:
```

Remove Breakpoint

The K command removes ("Kills") a breakpoint at a specific address if followed by an expression, or ALL (caution!) breakpoints if used alone.

```
DB: B
    1C00 4FD3 1277
DB: K 4FD3
DB: B
    1C00 1277
DB: K
DB: B

DB:
```

GO - RESUME PROGRAM EXECUTION

The G ("Go") command is used to resume program execution after a breakpoint. If a breakpoint exists at the present program counter address, that breakpoint is not inserted so that it is not immediately re-executed. A loop must have at least two breakpoints in it if execution is to be suspended each time through the loop.

Note that the "E" command is usually used before the first "G" command to set up the program to be tested. As mentioned previously, the Interactive Debugger initially sets up a default stack so the command:

```
G expression
```

can be used to start a program at the address the expression evaluates to.

Here are some examples:

```
DB: G 4C00
DB: G :PC+100
DB: G [.]
```

OS-9 RELATED COMMANDS

SHELL COMMAND

This command calls the shell to execute one or more system command lines. Its format is a dollar sign optionally followed by a shell command line. If the command line is given, the shell will execute just that line and return back to the debugger. If the dollar sign is immediately followed by an end-of-line, the shell will print prompts for one or more command lines in its usual manner. You can return to the (undisturbed) debugger by typing an end-of-file character (usually ESCAPE).

This command is useful for calling the system utility programs and the Interactive Assembler from within the debugger.

```
DB: $dir

DB: $unlink mypgm; mdir e; load test5

DB: $asm myprogram o=myprogram
```

QUIT COMMAND

This command (Q) causes the Interactive debugger to execute a EXIT system call which normally kills it and notifies its parent process. This generally returns you to the program that you were previously executing (typically the Shell).

Example:

```
DB: Q

OS9:
```

EXECUTE COMMAND

The Execute Command (E followed by a text line) performs the rough equivalent of the CHAIN system command, except instead of starting the new program (and overlaying the Interactive Debugger itself), it sets up its stack and all the debugger commands operate on the new program (G starts it). Note that this command will allocate program and data area memory as appropriate. The new program uses the debugger's current standard I/O paths, but can open other paths as necessary. In effect, the debugger and the program become coroutines.

This command is acknowleged by a register dump showing the program's initial register values. The "G" command is used to begin actual program execution.

The "E" command will set up the MPU registers as if you had just performed an F$CHAN service request as shown below:

```
high     +---------------+  <-- Y
         !               !
         !   parameter   !
         !     area      !
         !               !
         +---------------+  <-- X, SP
         !               !
         !               !
         !   data area   !
         !               !
         !               !
         +---------------+
         !  direct page  !
low      +---------------+  <-- U, DP

          D = Parameter area size
         PC = Module entry point absolute address
         CC = (F=0), (I=0)
```

Example:

```
         DB: E myprogram
                SP   CC  A  B DP  X     Y    PC
               0CF3 C8  00 01 0C 0CFF 0D00 9214
         DB:
```

LINK COMMAND

This command (L followed by text) attempts to link to the module who's name is given in the text line. If successful, Dot is set to the address of the first byte of the program and is displayed.

Example:

```
         DB: L FPMATH
             EC00 87
         DB:
```

USING THE DEBUGGER WITH A REAL PROGRAM

The example program shown below is presented here to show how the various debug commands may be used with a real program. This program prints "HELLO WORLD" and then waits for a line of input:

```
                             NAM                  EXAMPLE
                             OPT    O,-M
                             USE    /D0/DEFS/OS9DEFS

                    *
                    * OS-9 System Definition File Included
                    *

                             opt    1
0000                         ORG    0
0000                LINLEN   RMB    2             LINE LENGTH
0002                INPBUF   RMB    80            LINE INPUT BUFFER
0052                         RMB    150           HARDWARE STACK
00E7                STACK    EQU    .-1
00E8                DATMEM   EQU    .             DATA AREA MEMORY SIZE

0000 87CD0047                MOD    ENDPGM,NAME,$11,$81,ENTRY,DATMEM
000D 4558414D       NAME     FCS    /EXAMPLE/     MODULE NAME STRING

0014                ENTRY    EQU    *             MODULE ENTRY POINT
0014 308D0020                LEAX   OUTSTR,PCR    OUTPUT STRING ADDRESS
0018 108E000C                LDY    #STRLEN       GET STRING LENGTH
001C 8601                    LDA    #1            STANDARD OUTPUT PATH
001E 103F8C                  OS9    I$WRLN        WRITE THE LINE
0021 2512                    BCS    ERROR         BRA IF ANY ERRORS
0023 3042                    LEAX   INPBUF,U      ADDR OF INPUT BUFFER
0025 108E0050                LDY    #80           MAX OF 80 CHARACTERS
0029 8600                    LDA    #0            STANDARD INPUT PATH
002B 103F8B                  OS9    I$RDLN        READ THE LINE
002E 2505                    BCS    ERROR         BRA IF ANY I/O ERRORS
0030 109F00                  STY    LINLEN        SAVE THE LINE LENGTH
0033 C600                    LDB    #0            RETURN WITH NO ERRORS
0035 103F06         ERROR    OS9    F$EXIT        TERMINATE THE PROCESS
0038 48454C4C       OUTSTR   FCC    /HELLO WORLD/ OUTPUT STRING
0043 0D                      FCB    $0D           END OF LINE CHARACTER
000C                STRLEN   EQU    *-OUTSTR      STRING LENGTH
0044 268A06                  EMOD                 END OF MODULE
0047                ENDPGM   EQU    *             END OF PROGRAM
```

A Session With The Debugger

Below is an example of how DEBUG might be used on the program above. DEBUG is called from OS-9, the $ command is used to tell SHELL to load "EXAMPLE" into memory, the "L" command is used to link to it, etc.

```
OS9:DEBUG #2K

Interactive Debugger
DB: $LOAD /D1/EXAMPLE
DB: L EXAMPLE
   9200 87

DB: .
    9200 87
DB: M . .+44
9200 87CD 0047 000D 1181 9300 1400 E845 5841 ...D.........EXA
9210 4D50 4CC5 308D 0020 108E 000C 8601 103F MPL.0.. .......?
9220 8C25 1230 4210 8E00 5086 0010 3F8B 2505 .%.0B...P...?.%.
9230 109F 00C6 0010 3F06 4845 4C4C 4F20 574F ......?.HELLO WO
9240 524C 440D A484 7F8D D4A6 A02A F639 3432 RLD........*.942
DB: E EXAMPLE
      SP  CC  A  B DP  X    Y    U    PC
     0DF3 C8  00 01 0D 0DFF 0E00 0D00 9214
DB: .
    9200 87
DB: B .+2E
DB: G
HELLO WORLD
hello computer

    BKPT:
     SP  CC  A  B DP  X    Y    U    PC
    0DF3 C0  00 01 0D 0D02 000F 0D00 922E
DB: M :U   :U+20
0D00 FA31 6865 6C6C 6F20 636F 6D70 7574 6572 .lhello computer
0D10 0DDF C005 E9F1 95FA 4C0D 1DFA 0AC4 5900 ........L.....Y.
0D20 0B64 360B CFB1 0091 F820 5AE2 5AF8 5AF8 .d6...... Z.Z.Z.
DB: . :U+2
    0D02 68
DB:
    0D03 65
DB:
    0D04 6C
DB:
    0D05 6C
DB: Q

OS9:
```

INTERACTIVE DEBUGGER COMMAND SUMMARY

This Page Intentionally Left Blank

ERROR REPORTING

The Interactive Debugger will detect several types of errors which cause an error message and code number to be displayed. The error codes are always displayed in decimal notation. The various codes and descriptions are listed below. Error codes other than those listed are standard OS-9 error codes returned by various system calls.

0 ILLEGAL CONSTANT: The expression included a constant that had an illegal character or was too large (> 65535).

1 DIVIDE BY ZERO: A division was attempted using a divisor of zero.

2 MULTIPLICATION OVERFLOW: the product of the multiplication was greater then 65535

3 OPERAND MISSING: An operator was not followed by a legal operand.

4 RIGHT PARENTHESIS MISSSING: misnested parentheses

5 RIGHT BRACKET MISSING: misnested brackets

6 RIGHT CARAT MISSING: misnested byte-indirect (< and >)

7 INCORRECT REGISTER: misspelled, missing or illegal register name followed the colon.

8 BYTE OVERFLOW: attempted to store a value greater than 255 in a byte-sized destination.

9 COMMAND ERROR: misspelled, missing or illegal command.

10 NO CHANGE: the memory location did not match the value assigned to it.

11 BREAKPOINT TABLE FULL: the maximum number of twelve breakpoints already exist.

12 BREAKPOINT NOT FOUND: no breakpoint exists at the address given.

13 ILLEGAL SWI: A SWI instruction was encountered in the user program at an address other than a breakpoint.

This Page Intentionally Left Blank