D-Bug4744 Monitor User Manual

Licensing Agreement

Table of Contents

1. General Instructions (good for all monitor commands)

1.1 Entering Numbers

D-Bug4744 only understands numbers entered in hex. For example, to enter the number 123, use the hex equivalent: 7B.

All addresses – designated by “<addr>” – must must be entered as 4 hex characters. For example, the memory display function (md <addr> <addr>) can be used correctly with the following two commands:

```
md 0800 0810

md e000 e0ff
```

The following two commands would be incorrect:

```
md 800 810

md 1 ff
```

All byte-size number inputs – designated by “<val>” – must be entered as two characters. For example, the register modify (rm <reg> <val>) on a byte sized register (a,b) should be used like this:

```
rm a 05
```

It would be incorrect to enter that command like this:

```
rm a 5
```

Incorrect number entry will cause inexpected and guaranteed incorrect results!

1.2 Register Names

The register view (*rv*) and register modify (*rm*) functions use register names for display and to allow the user to edit the internals of the registers. D-Bug4744 only recognizes the following register names: a, b, x, y, s, c. They must be entered in lower-case. The “D” register is not recognized.

Name	*Register*	*Name*	*Register*
a	A	b	B
x	X	y	Y
s	SP	c	CCR

Table 1: Register Names

2. Commands

2.1 Memory Display [md]

Syntax	`md <addr_low> <addr_high>`
Description	Displays 16-byte lines of memory between *addr_low* and *addr_high*. The two limits – *addr_low* and *addr_high* – are rounded down and up, respectively, to the nearest 16-byte boundaries. For example, 0x0801 as a low bound would be rounded down to 0x0800.

2.2 Memory Modify [mm]

Syntax	`mm <addr> <val>`
Description	Changes the memory value at *addr* with *val*. It also displays the former value of that memory location (just prior to changing it.)

2.3 Register View [rv]

Syntax	`rv`
Description	Displays the contents of the registers. Note that it does not display the contents of the registers as used by the monitor. It “unstacks” the register values from the user's stack and displays those values. This implies that the user has run code which has been interrupted (by an SWI, for example) and the processor has therefore put the state of the user's code onto the user's stack. The rv command then reads and displays the state of the user's code. This is done to facilitate ease of debugging user code. (If the user's stack doesn't have anything on it, this command will display random values for the registers.)

2.4 Register Modify [rm]

Syntax	`rm <reg> <val>`
Description	Changes the value contained in the specified register. (See Sect. 1.2 for a list of register names.) As with the register view command, this command does not modify the state of the current registers but rather modifies the state of the user's registers on the stack. If there are no user registers on the stack, it will modify some random memory location so beware.

2.5 Block Fill [fill]

Syntax	`fill <addr_low> <addr_high> <val>`
Description	Fills the memory between *addr_low* and *addr_high* with the byte in *val*. Does not round the addresses.

2.6 Go [go]

Syntax	`go <addr>`
Description	Does a JMP to *addr*. Switches to the user's stack before the JMP.

2.7 S19 Load [load]

Syntax	`load`
Description	Loads an S19 from the SCI port (from the host) to memory. It will not write to FLASH or EEPROM, only RAM. This command returns to the monitor code when an “S9” record is received. If there is no S9 record, it will hang.

2.7 Resume [res]

Syntax	`res`
Description	Resumes execution of user code after an SWI (user supplied or breakpoint.)

2.8 Breakpoint [break]

Syntax	`break` : prints the breakpoint table `break + <addr>` : adds a breakpoint at addr `break - <addr>` : removes the breakpoint at addr
Description	Adds, removes and prints the breakpoint table.

2.9 Disassemble [dasm]

Syntax	`dasm <addr> <val>`
Description	Disassembles *val* number of instructions beginning at *addr*. If the disassembler is unable to disassemble an instruction, it fails with the message “incomplete disassembly...” Some addressing modes do not disassemble at all. See Section 3.2 for more information.

3. D-Bug4744 Internals

3.1 User and Monitor Stacks

D-Bug4744 maintains two stacks. When user code is executed (using the *go* command, for example) the monitor switches stack pointers to the user's stack. When control of the processor returns to the monitor, the stack pointer is changed back to the monitor's stack to prevent corruption of the user's program state.

The *go* and *resume* commands do this changing of the stack pointer. When an SWI occurs (and the monitor SWI handler is called) the stacks are again switched.

The register view and register modify commands read the register values off the user's stack. If the user has not executed any code (when the board is reset straight into the monitor, for example) reading the user's stack will show random values because it has no way of knowing if the user's stack has legitimate information on it.

3.2 Disassembly

The built-in disassembler does not correctly disassemble all addressing modes for some instructions. Nor does it disassemble all instructions. The general instructions (such as ldaa, ldab, ldx, ldy for example) have four “base” addressing modes: immediate, direct, extended and indexed. Each of these corresponds to a particular opcode when code is assembled. The last mode – indexed – contains another five different modes:

Mode	Description
IDX	5-bit constant offset addressing. Example: ldaa -1,x
IDX1	9-bit constant offset addressing. Example: ldaa -254,x
IDX2	16-bit constant offset addressing. See below.
[D,IDX]	Accumulator D indirect offset addressing. Example: ldaa [D,x]
[IDX2]	16-bit constant indirect indexed addressing. Example: ldaa [10,x]

Table 2: Indexed Addressing Modes

The IDX2 mode – 16-bit constant offset addressing – didn't work in the assembler I was using to test the disassembler so it may or may not work.

The only modes that the disassembler recognizes are immediate, direct, extended and the IDX and IDX1 indexed modes. The disassembler has the ability to recognize the other modes but currently the monitor doesn't decode them.

The instruction opcodes that the disassembler specifically cannot recognize are given in the following table.

dbeq	emaxm	emuls	ibne	leay	minm
dbne	emind	etbl	jmp	maxa	mem
emacs	eminm	exg	leas	maxm	movb
emaxd	emul	ibeq	leax	mina	movw
Rev	Revw	sex	tfr	tbeq	tbne
tbl	tpa	tsx	tsy	txs	tys
wav	xgdx	xgdy			

Table 3: Unsupported Instructions

3.3 Interrupt Vectors

In order to allow users to change the interrupt vectors, pseudo-vectors have been placed at the beginning of internal RAM (0x0800). All of the hard-coded vectors at the top of program space (0xffce-0xffff) except SWI and RESET point to these pseudo-vectors in RAM. Each pseudo-vector is three bytes to accommodate a JMP instruction. If a user wants to redirect the SCI0 interrupt, for example, they would put a JMP to the interrupt handler at 0x080c. The following table gives the locations for the pseudo-interrupts.

Interrupt	RAM Addr	Interrupt	RAM Addr
KEY WKUP J	0800	TIMER 3	0827
KEY WKUP H	0803	TIMER 2	082a
A/D CONV	0806	TIMER 1	082d
SCI1	0809	TIMER 0	0830
SCI0	080c	RTI	0833
SPI TX COMPLETE	080f	IRQ	0836
PAC INPUT EDGE	0812	XIRQ	0839
PAC OVERFLOW	0815	SWI	083c
TIMER OVERFLO	0818	TRAP	083f
TIMER 7	081b	COP FAIL	0842
TIMER 6	081e	COP CLOCK FAIL	0845
TIMER 5	0821	RESET	0848
TIMER 4	0824		

Table 4: Pseudo-Interrupt Vectors

The SWI and RESET vectors are used by the monitor and are therefore hard-coded into the ROM but space for a pseudo-vector is left for future use.

3.4 Breakpoints and How They Work

Breakpoints are implemented in D-Bug4744 by replacing user code at the specified breakpoint address with an SWI (software interrupt.) For that reason, breakpoints must be placed on the first byte of an instruction if it is a multi-byte instruction. (This makes sense also because if you put a breakpoint on the second byte, for example, the PC would never reach that byte – the PC only increments over whole instructions.) They can't be placed on jump or branch instructions.

Internally, when a breakpoint is reached, the monitor removes that SWI and replaces it with the code that was originally there. It then puts another SWI at the first byte of the next instruction following this one. When you resume after a breakpoint, it resumes at the location of the instruction that was overwritten. Because the next instruction is now an SWI, it goes back to the monitor and replaces the breakpoint SWI again.

Because the monitor uses its disassembler to find the beginning of the next instruction, breakpoints can only be placed on instructions that the disassembler recognizes.

There is another way to use breakpoints and that is to manually add SWI instructions to your code. When the SWI is reached, the monitor is called and will allow you to do everything you would if there were a breakpoint at that location. This is arguably the easiest way to debug your code.

4. Downloading Code

The *load* command is used to get code into RAM. When executed, it waits for an S19 file to be received through the SCI port.

The downloading ends when an S9 record (usually the last record in the file) is received. If an S9 record is not received, the monitor will hang. Resetting the board will usually solve this problem without corrupting the downloaded data in RAM.

This downloader will not load code into either FLASH or EEPROM. Any attempts to write to those locations will result in failure.

```
S0030000FC
S113FFCE08030800080608090080C080F081208158B
S113FFDE0818081B081E08210824082708 2A082DBB
S113FFEE0830083308360839E194083F08420845BA
S105FFFEE0001D
S104EF603D6F
S9030000FC
```

Table 5: Sample S19 File

S105FFFEE0001D	
S1	Designator (tells the “type” of the line) S0,S9 = “administrative lines” S1 = data line
05	Number of bytes (including two for address, one for checksum) in this line
FFFE	Address where this line of bytes goes
E0 00	Data to be placed at the above addresss
1D	Checksum (you'll have to find a detailed document for how this is computed). It can usually be ignored and is ignored by D-Bug4744.

Table 6: Decomposed S19 Line

5. Debugging Tips

5.1 Use Breakpoints Wisely

Breakpoints can either be extremely helpful or unnecessary to the debugging effort. In general, the easiest way to use breakpoints with D-Bug4744 is to add SWI instructions at critical points in the program's execution. The easiest way to do this is not with the built-in breakpoints but by assembling them into the code by hand. This is a fast, simple way of utilizing the capabilities of the monitor to view the state of the system at a given place in the code using the register and memory view/modify routines.

It is better to reserve use of the monitor's internal breakpoints for situations where you don't have easy access to an assembler to add your own.

5.2 Print Debugging Information

The D-Bug4744 monitor has general purpose subroutines for printing strings, hex words (4 characters), hex bytes (2 characters) and simple characters to the SCI. These subroutines can be called from the user code by doing a jsr to the proper place in memory. (You can find the locations of all of these functions in the subroutine table in Section 6.) Using these functions for debugging can be as simple as printing a character after each subroutine call in a piece of code to see which routine is causing the system to halt. A common practice is to print letters of the alphabet ('a', 'b', 'c', etc) such that if the last character printed is 'd', then you know that the subroutine coming after 'd' is causing problems.

The printing subroutines in D-Bug4744 makes it easy to print addresses to determine if a *jmp* is going off-target, for example. You can also print values in registers before comparisons, values in memory and more.

5.3 Start Small

When writing large assembly programs, it is best to start with small, manageable subroutines that you can test independently of any other code. In fact, it is sometimes useful to write these subroutines and code to test them in completely separate files. Then, when all of the subroutines are tested, combine them into the final product. (Use of the *#include* assembler directive can be helpful here.)

6. Utility Subroutines

The following D-Bug4744 subroutines can be called by user code.

Function	print_string
Address	***ef27***
Inputs	Y : pointer to null-terminated string
Outputs	None
Modifies	A, Y
Description	Prints a null (0x00) terminated string to the SCI port

Function	print_address
Address	***ef4f***
Inputs	X : address to print
Outputs	None
Modifies	None
Description	Prints an address (ex: 4faa) in hex to the SCI port

Function	print_number
Address	***ef33***
Inputs	A : Number to print
Outputs	None
Modifies	None
Description	Prints a number (ex: 4f) in hex to the SCI port

Function	put_char
Address	***ef20***
Inputs	A : character to print
Outputs	None
Modifies	A
Description	Sends a character to the SCI port

Function	get_address
Address	***e7e7***
Inputs	None
Outputs	X : Address
Modifies	None
Description	Waits for a 4-character hex address (ex: 43ff) from the SCI port and returns it in X

Function	get_char
Address	***ef03***
Inputs	None
Outputs	A : Character from SCI port
Modifies	A
Description	Returns a character from the SCI port if one is available – it doesn't block.

Function	get_char_wait
Address	***ef0c***
Inputs	None
Outputs	A : Character from SCI port
Modifies	A
Description	Returns a character from the SCI port if one is available – it does block (it waits for a character from the SCI port before it returns.)

Function	get_line
Address	***e884***
Inputs	X : Pointer to character buffer (where line will be stored) B : Total number of bytes to return (limits line length)
Outputs	None
Modifies	None
Description	Grabs characters from the SCI port and fills the character buffer until a 0x13 (linefeed) is received or until it receives the number of bytes given in register B. This function does NOT null-terminate the string it receives.

6.1 Example Code for Calling D-Bug4744 Subroutines

The following snippet shows how to call built-in D-Bug4744 subroutines from within user code.

```
print_address      equ      $ef4f
put_char           equ      $ef20
print_number       equ      $ef33
print_string       equ      $ef27

          org      $d000
string    db.c     'This is a test: ',0
newline   db.c     13,10,0

test_code_begin:
          ldy      #string
          jsr      print_string
          ldx      #$1234
          jsr      print_address
test_code_end      equ      *
```

That code produces the following output at the SCI port:

```
This is a test: 1234
```

One helpful string is the “newline” string that sends a CRLF (carriage return, line feed) to an attached console (Minicom, Hyperterminal, etc.)

A. About D-Bug4744 and the Motorola 68HC12B32

A.1 Origin of D-Bug4744

This section has been removed. See the comments in the D-Bug 4744 source code for more information.

Contact the author if you're really curious.

A.2 The Motorola 68HC12B32

The argument goes that the 6812 is better than the 6811. It isn't. In fact, the 6812 is the biggest hack of a processor that I've ever seen. (And I don't use *hack* in a good sense.) It is poorly designed and poorly documented. The development tools are substandard by any comparison with their competitors. It has functionality (“fuzzy logic instructions”) that demonstrates the same misunderstanding of the market that led to the Iridium disaster. They offer only one package type – a very non-standard 80-pin TQFP. Its slow by comparison to other similarly-priced microcontrollers. It can only be programmed by using a special tool or one of Motorola's eval boards (and some software that we wrote for the purpose.) The interface to external memory is a terrible non-standard clock-stretching scheme that doesn't quite work the way an ordinary engineer would make it work. (In fact, it can be argued that it simply doesn't work, period.)

Processors that the 68HC12B32 is inferior to: Atmel AVR, Microchip PIC, AMD186, IBM PowerPC 4xx and Motorola ColdFire.

I think Motorola needs to wake up and take a look at what Atmel and Microchip and a slew of other companies are doing with their processor designs. They need to give out (free) better dev tools like compilers, assemblers, simulators and downloaders. Or at least support an open-source development of those tools. They should remove the “special” instructions, get more flavors of the chip going in more packaging options, make programming the device easier and make it faster. But I think that as long as Motorola sticks to the CISC architecture, not too many changes will be made.

B. Contact Info

If you want to report bugs in this code, please send email to:

bugs4744@mil.ufl.edu

For non-technical questions (like more information on how *not* to design a microprocessors class) you *could* send email to me personally at

pomalley@mil.ufl.edu

I say *could* because you may never hear from me. When I'm busy, I can withstand to have a million emails in my inbox without worrying about replying.

The lab where D-Bug4744 was written and the hardware for the microprocessors class was designed is the Machine Intelligence Lab at the University of Florida. You can check out what we do at

http://www.mil.ufl.edu

The microprocessors class (EEL4744) is usually hosted off the MIL website under “courses”.

We usually design robots and things like that at the Lab but through doing that, we've also put together a good group of hardware designers. And we can write code, too. (Well, *we* think we can.)

My website is at

http://www.mil.ufl.edu/~pomalley

Enjoy!