Dbmon Reference Manual

September, 1983

1. Introduction

This document describes National Semiconductor's monitor, *dbmon. Dbmon* is designed to be used in a NS16000™–based microcomputer. It is tailored for use in remote debugging of the microcomputer with *ddt*(1) running on a host system. (*Ddt* is available as part of the GENIX™ Cross-Support package for a VAX[1] running UNIX[1], or with GENIX running on an NS16000 based system.)

In addition to running with *ddt*, *dbmon* can also be used directly either from a terminal connected to the target system or from a terminal connected to the host, but with the program *cu16*(1) (see also: GENIX Cross-Support Software for an NS16000-based System) managing the serial line between the host and the target system. Since the monitor commands are tailored for use with *ddt*, some of the commands may be inconvenient when entered directly by the user.

A typical application of the monitor would be in a DB16000 board connected to a VAX by a serial line. Information specific to this application is included in this document. For this application, *dbmon* replaces the monitor supplied with the DB16000. It provides functions for debugging similar to those supplied by other DB16000 monitors. Except for loading and executing programs, *dbmon* does not provide operating system functions.

2. Notation Conventions

Certain characters are specified symbolically in this document:

$	= escape
<cr>	= return
<lf>	= line feed (control/j)
<bs>	= backspace

3. Configuration

Dbmon's actions may depend upon the setting of hardware configuration switches on the target board. The symbol *options* in the monitor source listing file *rom.S* is used to store the address of the location the monitor checks to determine the state of the option switches.

The "set" position of a switch is the logical 1 or true position.

Switch 2 is the only one significant switch on the DB16000 board. If set, typed characters will be echoed, and an <lf> will be echoed after each <cr>. If not set, characters are not echoed.

Both *ddt* and *cu16* require that switch 2 be set.

Switch 2 is set if the switch is in the "open" or "off" position. (Switch 2 is one of the subswitches of switch 3, the baud rate/ configuration switch.)

4. Running Mode

The monitor runs in supervisor mode. By default, a user program loaded with *dbmon* starts in supervisor mode, though with the stack pointer set to SP1 (user stack pointer). However, the program may be started in user mode by first changing the setting of the PSR with the command "cpsr=0b00".

GENIX, and NS16000 are trademarks of National Semiconductor Corporation.

[1] VAX is a trademark of Digital Equipment Corporation

[2] UNIX is a trademark of Bell Laboratories.

GENIX was derived from 4.1bsd which was developed at the University of California at Berkeley.

The information in this document is for reference only and is subject to change without notice.

5. Command Input Protocol

The monitor reads lines up to a <cr>. Commands are not executed until the <cr> is typed. <lf> characters are ignored.

The backspace character may be used to delete previously typed input. The escape key will abort commands at any time. It will also discard any input and terminate a control/q wait.

All numeric quantities are entered in hex. Spaces may be inserted before any number.

Commands and hex numbers may be typed in upper or lower case.

The monitor's output may be suspended with control/s and resumed with control/q.

6. Memory Organization

Dbmon requires no RAM except for space on the interrupt stack. The interrupt stack begins at 64k on the DB16000.

A program which moves the interrupt stack may still reenter *dbmon*; it does not matter where the stack is in RAM when the monitor is entered.

Initial Memory Organization

System	Area	Low	High
DB16000	Interrupt Stack	˜ 63.8k	64k
DB16000	User Stack	N.A.	60k

The monitor runs in supervisor mode and always uses the interrupt stack. A program can run in either mode with either stack. There are two reasonable stack configurations for the monitor and program:

(1) Program uses SP1 and separate memory locations are reserved for each stack.

(2) Program uses SP0 (program and monitor run on the same stack).

In the first case, the program must avoid the memory used by the interrupt stack, because those locations will be overwritten when the monitor is entered.

In the second case, the program can allocate memory anywhere; however, if the program fails because its stack is not in RAM, the monitor will also fail.

The first case is *dbmon's* initial organization. Specify "cps=0" to change to case two.

7. *Dbmon* Commands

In the following command descriptions, lower-case letters represent literal character input and upper-case letters are used to indicate parameters, while either case is actually allowed in the command.

The commands groups are as follows:

- memory and register print commands
- memory and register change commands
- read and write MMU registers commands
- set configuration register command
- group data commands, e.g., move, fill and load blocks of memory

- program execution control commands
- additional commands
- test commands
- reserved commands

Note: The print and change virtual-memory commands require an NS16082 MMU chip.

7.1. Print Commands

The contents of memory and the registers may be printed with the following commands:

all	- Print all the registers.
ppsr	- Print the processor status register.
psb	- Print the static base register.
pis	- Print the interrupt stack pointer (SP0).
pmod	- Print the mod register.
pfp	- Print the frame pointer.
pus	- Print the user stack pointer (SP1).
psp	- Print the user stack pointer (SP1). Note: **pus** and **psp** are functionally identical.
pintbase	- Print the interrupt base register.
prN	- Print general register N.
ppc	- Print the program counter.
pmbA	- Print one byte of memory at address A.
pmwA	- Print one word of memory at address A.
pmdA	- Print one double-word of memory at address A.
pvbA	- Print one byte of memory at virtual address A.
pvwA	- Print one word of memory at virtual address A.
pvdA	- Print one double-word of memory at virtual address A.

Note that the “**p**” commands are constructed in the following way: the letter **p** followed by a register name, e.g., **fp** or **r1** or **m** for physical memory, or **v** for virtual memory. The **m** or **v** is then followed by a **b** , **w** ,or **d** for byte, word or double-word and then the address of interest.

All register names may be abbreviated to two characters, e.g., **in** for **intbase.**

7.2. Change Commands

Memory and registers may be changed with the following commands:

cpsr=V	- Change the processor status register to value V.
cmod=V	- Change the mod register to value V.
cfp=V	- Change the frame pointer to value V.
cus=V	- Change the user stack pointer (SP1) to value V.
csp=V	- Change the user stack pointer (SP1) to value V. Note: **cus** and **csp** are functionally identical.
cintbase=V	- Change the interrupt base register to value V.
crN=V	- Change general register N to value V.
cpc=V	- Change the program counter to value V.
cmbA=V	- Change one byte of memory at address A to value V.
cmwA=V	- Change one word of memory at address A to value V.
cmdA=V	- Change one double-word of memory at address A to value V.
cvbA=V	- Change one byte of memory at virtual address A to value V.
cvwA=V	- Change one word of memory at virtual address A to value V.
cvdA=V	- Change one double-word of memory at virtual address A to value V.

Note that the “**c**” commands are constructed in the same manner as the “**p**” commands.

All register names may be abbreviated to two characters, e.g., “**mo**” for “**mod**”.

There is no change register command for the interrupt stack or static base registers. The interrupt stack pointer can be changed with a supervisor mode program; the static base register can be changed by altering the module entry selected by the mod register.

7.3. MMU Commands

The MMU registers are read from, and written to, with:

wN=V	- Write value V to MMU register N.
rN	- Read (and print) MMU register N (the registers and the correspnding value of N: bpr0=0 bpr1=1 pf0=4 pf1=5 sc=8 msr=a bcnt=b ptb0=c ptb1=d eia=f)

7.4. Configuration Register Command

The configuration can be changed with the “**x**” command:

xV	- execute a setcfg instruction for configuration V (icu=1 fpu=2 mmu=4)

7.5. Data Commands

m A1 A2 N	- Move N bytes from A1 to A2.
m A1 A2 N [bwd]	- Move N bytes, words, or double-words from A1 to A2.
f A1 A2 DD	- Fill memory from A1 to A2 with data byte DD.
f A1 A2 V [bwd]	- Fill memory from A1 to A2 with byte, word, or double-word value V.

d A1 N - Dump (print) **N** bytes of memory starting at **A1**. The format for the output of this command is the same as the format for input to the load command.

Note: Although the data is printed in 8-digit groups, these groups are not double-words, as the order of the bytes is from the least significant to the most significant which is the reverse of the order for double-words.

l A DDDDDDCC - Load bytes DD... starting at address A. Bytes are represented by pairs of hex digits, up to 16 data bytes (32 hex characters) per line. The line is terminated by a check sum byte, which is the 2-digit hex representation of the 8-bit sum of the data bytes on the line. If the transmitted checksum does not match the computed checksum, the message **E CRC** is printed with the value of the computed checksum.

image - Start the binary image loader. The image loader reads a header, data bytes, and then a 1-byte checksum from the serial line. The header consists of a 32-bit starting address and a 32-bit length value. These parameters are sent as 8-bit bytes, least significant byte first. The length parameter specifies the number of data bytes, to be transferred as 8-bit binary bytes. The check sum is also transferred as an 8-bit byte. <Esc>, <bs>, and control/s have no special meaning in this mode.

If “**i**” is typed from the keyboard, the DB16000 will probably need to be reset. Entering ten null bytes for an address, length, and checksum of zero may prevent the need for resetting the board, but only if the terminal uses even parity.

7.6. Program Execution Commands

s	- Single-step: execute the instruction at the current pc.
g	- Start the program (with the current pc).
q	- Do a nonsequential fetch step, using NS16082 MMU.

7.7. Additional Commands

v A1 L	- Compute and print the 32-bit checksum of the **L** bytes beginning at address **A1**
!	- Ignore the remainder of a line. Useful for comments when *dbmon* commands are stored in a file.

7.8. Test Commands

t1	- Run test 1. This test probes the RAM configuration at every 4k interval, up to 2^20 bytes. The test reports the first and last address of available memory and possibly the last address checked.
t2 A N	- Run test 2 starting at address A for N bytes. This test runs the memory diagnostics described in Section 9.

8. NS16082 MMU Dependencies

There are two versions of *dbmon* supplied in PROM. The first version, *dbmon1*, is for use in systems with an NS16082 MMU; the second version, *dbmon2*, is for use in systems without an MMU.

The monitor program source file, **rom.S**, contains the source for the version which uses the MMU. The **rom.S** file can be modified to produce a custom monitor. If the custom monitor is not to be dependent on an MMU, a search for the name **NOMMU** in the source file will find comments indicating lines which need to be deleted for non-MMU systems. Do not use the commands **pv[b,w,d], cv[b,v,d], q, r#,** and **w#** when running a monitor built without those source lines. These commands use the MMU slave instructions and will cause an undefined trap in a program modified for no MMU.

9. Memory Testing

The “**t2**” command starts test two, the memory test sequence. Eight subtests are run in sequence; the sequence is restarted after the last subtest. The tests destroy the original contents of the addresses checked. The memory test itself uses no RAM; however, if a failure is found, RAM locations on the stack are used temporarily by the formatted output routines. If an error occurs and the stack is within the memory range being checked, spurious errors may be reported in addition to any actual errors. The first error reported will always be a real error.

9.1. Test Patterns

Each of the subtests cycles through seven patterns (see Table 1).

Table 1. Memory Test Patterns In Hexidecimal

No.	Pattern	Function
1.	00000000	all zeros
2.	ffffffff	all ones
3.	aaaaaaaa	adjacent bits different
4.	55555555	adjacent bits different
5.	11224488	a different high in each byte
6.	eeddbb77	a different low in each byte
7.	39a7c736	miscellaneous bits

9.2. Test Sequence

Each subtest prints its name as it starts (see Table 2).

Table 2. Memory Tests

No.	Name	Context	Test
1.	parity	words	parity(address) xor pattern
2.	addr ->	double-words	address xor pattern, low to high
3.	addr <-	double-words	address xor pattern, high to low
4.	doubles	double-words	pattern
5.	e. words	even words	pattern
6.	e. bytes	even bytes	pattern
7.	load	doubles	high transfer rate

9.3. Test Failures

When a test fails, a message is printed with the following format:

Failure(=addr=xor=test=mem)=AAAA=XXXX=TTTT=MMMM

AAAA
- address containing incorrect data.

XXXX
- pattern used to provide the current test variation.

TTTT
- expected contents of memory.

MMMM
- actual contents of memory.

9.4. Memory Test 1: Address Parity

This test computes even parity for each word address. The parity bit is then extended to form 16 bits of zeros or ones; and the exclusive-or of this value and the current pattern is written to the word.

The words are checked on a second pass.

9.5. Memory Test 2: Increasing Addresses

For each double-word address, this test writes the exclusive-or of the pattern and the address to the double-word. The addresses are written to in increasing order from the base address. They are checked on a second pass.

9.6. Memory Test 3: Decreasing Addresses

This test is similar to Memory Test 2, but the addresses are written to in decreasing order of address, starting from the base address. The addresses are checked, in increasing order, on a second pass.

9.7. Memory Test 4: Double Words

Test 4 writes each pattern into each double-word.

9.8. Memory Test 5: Even Words

The lower word of each pattern is written to each even word.

9.9. Memory Test 6: Even Bytes

The lower byte of each pattern is written to each even byte.

9.10. Memory Test 7: Load

A sequence of double-word string moves are generated in order to test the memory system with a constant high transfer rate. This test uses memory parity hardware to detect errors. If the target board does not have memory parity hardware, the test is not useful.

Dbmon Reference Manual

Table of Contents