Using the dBUG Monitor Firmware on the MPC562BC Board

Rev. 0.1, 10/2002

The MPC562BC single board computer has a resident firmware package that provides a self-contained programming and operating environment. The firmware, named dBUG, provides the user with monitor/debug interface, inline assembler and disassembly, program download, register and memory manipulation, and I/O control functions. This document is a how-to-use description of the dBUG package, including the user interface and command structure.

1.0 What Is dBUG?

dBUG is a traditional ROM monitor/debugger that offers a comfortable and intuitive command line interface that can be used to download and execute code. It contains all the primary features needed in a debugger to create a useful debugging environment.

The firmware (stored in the upper 1MByte of the Flash ROM device) provides a self-contained programming and operating environment. dBUG interacts with the user through pre-defined commands that are entered via the terminal. These commands are defined in Section 2.4, “Commands”.

The user interface to dBUG is the command line. A number of features have been implemented to achieve an easy and intuitive command line interface.

dBUG assumes that an 80x24 character dumb-terminal is utilized to connect to the debugger. For serial communications, dBUG requires eight data bits, no parity, and one stop bit, 8N1 with no flow control. Xon/Xoff flow control should be turned on for downloading data to the board using the DL and DLDBUG commands. The default baud rate is 19200 but can be changed after the power-up.

The command line prompt is “dBUG> “. Any dBUG command may be entered from this prompt. dBUG does not allow command lines to exceed 80 characters. Wherever possible, dBUG displays data in 80 columns or less. dBUG echoes each character as it is typed, eliminating the need for any “local echo” on the terminal side.

In general, dBUG is not case sensitive. Commands may be entered either in upper or lower case, depending upon the user’s equipment and preference. Only symbol names require that the exact case be used.

Most commands can be recognized by using an abbreviated name. For instance, entering “he” is the same as entering “help”. Thus, it is not necessary to type the entire command name.

The commands DI, GO, MD, STEP and TRACE are used repeatedly when debugging. dBUG recognizes this and allows for repeated execution of these commands with minimal typing. After a command is entered, simply press <RETURN> or <ENTER> to invoke the command again. The command is executed as if no command line parameters were provided.

An additional function called the "System Call" allows the user program to utilize various routines within dBUG. The System Call is discussed at the end of this chapter.

The operational mode of dBUG is demonstrated in Figure 1. After the system initialization, the board waits for a command-line input from the user terminal. When a proper command is entered, the operation continues in one of the two basic modes. If the command causes execution of the user program, the dBUG firmware may or may not be re-entered, at the discretion of the user’s program. For the alternate case, the command will be executed under control of the dBUG firmware, and after command completion, the system returns to command entry mode.

During command execution, additional user input may be required depending on the command function.

For commands that accept an optional <width> to modify the memory access size, the valid values are:

- B 8-bit (byte) access
- H 16-bit (half-word) access
- W 32-bit (word) access

When no <width> option is provided, the default width is .W, 32-bit.

The core MPC500 register set is maintained by dBUG. These are listed below:

- GPR0-GPR31
- IP (SRR0 is IP)
- MSR (SRR1 is MSR)
- CR, XER, LR, CTR, DSISR, DAR, DEC

All control registers on MPC500 core are not readable by the supervisor-programming model, and thus not accessible via dBUG. User code may change these registers, but caution must be exercised as changes may render dBUG inoperable.

A reference to “SP” (stack pointer) actually refers to general purpose address register one, “GPR1."

2.0 Operational Procedure

2.1 System Power-up

- Be sure the power supply is connected properly prior to power-up.
- Make sure the terminal is connected to the RS232 DB-9 connector.
- Make sure the IP bit is set (CFG1 - this is hardwired by default). This will cause the board to boot out of external flash (where the dBUG code resides).
- Turn power on to the board.

Figure 1 shows the dUBG operational mode.

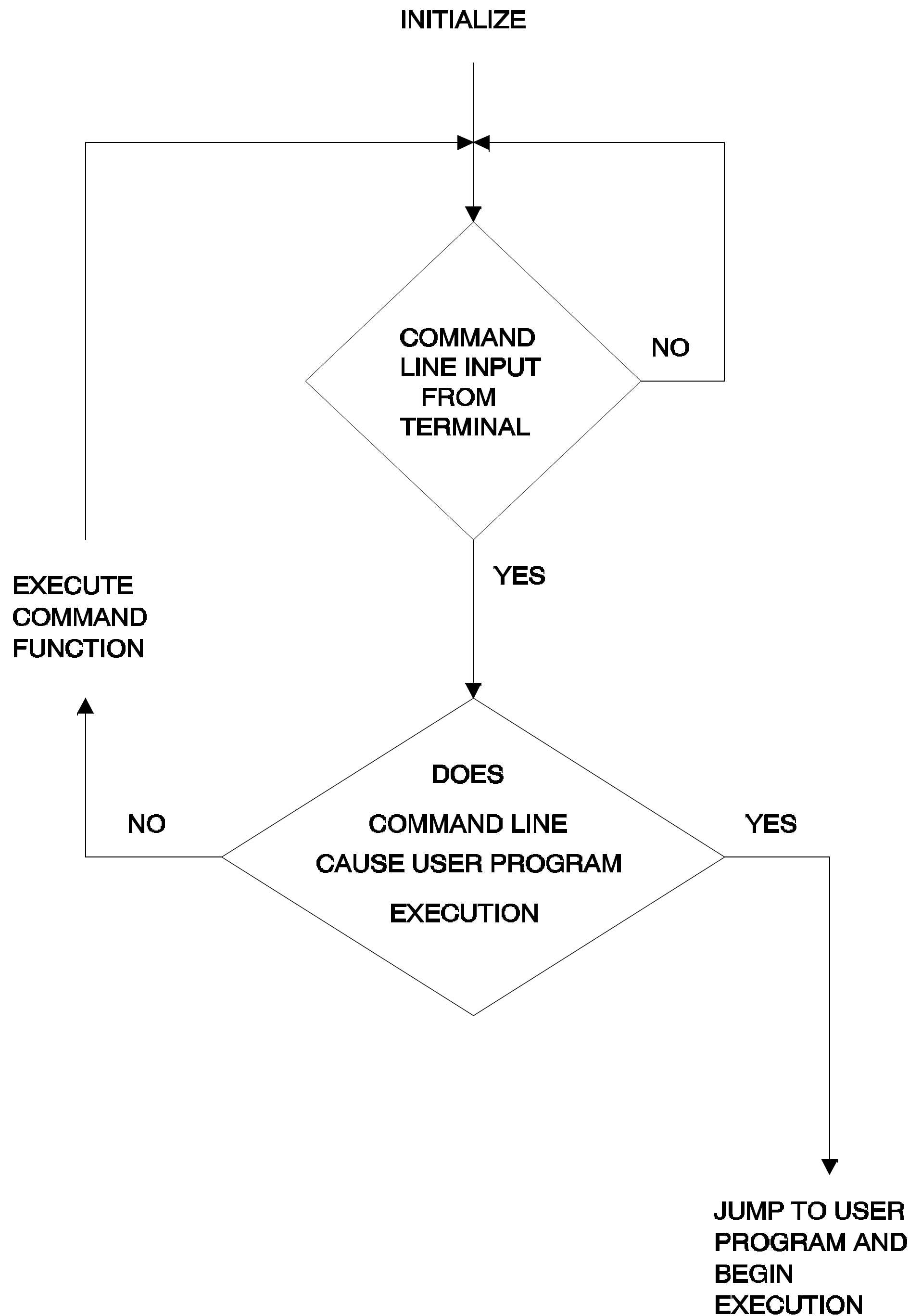


Figure 1. Flow Diagram of dBUG Operational Mode.

2.2 System Initialization

The act of powering up the board will initialize the system. The processor is reset and dBUG is invoked.

dBUG performs the following configurations of internal resources during the initialization. The IP bit is set by default, placing the vector table at 0xFFF0_0000 (external Flash). To take over an exception vector, the

Motorola Confidential Proprietary

user places the address of the exception handler in the appropriate vector in the vector table located at 0xFF00_0000. dBUG allows users to write their own exception handlers for the following exceptions: External Interrupt, Alignment, Program Floating-point unavailable, Decrementer, and Floating-point assist. dBUG will look in the User Exception Space (0xFF00_0000 - 0xFF00_1FFF) to see if there is user code there. If there is, it will execute the user's exception handler, if there is not (memory is erased to be 0xFFFF_FFFF), dBUG will execute it's own exception handlers (located in Flash at 0xFFF0_0000 - 0xFFF0_1FFF).

The Software Watchdog Timer is disabled and internal timers are placed in a stop condition. Interrupt controller registers are initialized with unique interrupt level/priority pairs. Please refer to the dBUG source files on the PowerPC website (www.motorola.com/powerpc) for the complete initialization code sequence.

After initialization, the terminal will display:

```
Part Number: 0x35
MaskNum: 0x20

Copyright 1995-2003 Motorola, Inc.  All Rights Reserved.
MPC562 MPC562BC Firmware v3b.1a.1a (Build 1 on Oct 28 2002 08:54:53)

Enter 'help' for help.


dBUG>
```

If you did not get this response check the setup, refer to Section 2.1, “System Power-up”.

Other means can be used to re-initialize the MPC562BC Computer Board firmware. These means are discussed in the following paragraphs.

2.2.1 Hard RESET Button

Pressing the Hard RESET button (HARD_RESET) causes all processes to terminate, resets the MPC562 processor and board logic and restarts the dBUG firmware. Pressing the HARD_RESET button would be the appropriate action if all else fails.

2.2.2 Non-Maskable Interrupt Button

SWITCH1 can be used as a non-maskable interrupt (NMI) button. It is available for the user to use in their code as an input if the appropriate jumper on JP3 is removed. The NMI function causes an interrupt of the present processing (a level 0 interrupt on MPC562) and gives control to the dBUG firmware. This action differs from RESET in that no processor register or memory contents are changed, the processor and peripherals are not reset, and dBUG is not restarted. Also, in response to depressing the NMI button, the contents of the MPC562 core internal registers are displayed.

The NMI function is most appropriate when software is being debugged. The user can interrupt the processor without destroying the present state of the system. This is accomplished by forcing a non-maskable interrupt that will call a dBUG routine that will save the current state of the registers to shadow registers in the monitor for display to the user. The user will be returned to the ROM monitor prompt after exception handling.

2.2.3 Software Reset Command

dBUG does have a command that causes dBUG to restart as if a hardware reset was invoked. The command is "RESET".

2.3 Command Line Usage

The user interface to dBUG is the command line. A number of features have been implemented to achieve an easy and intuitive command line interface.

dBUG assumes that an 80x24 ASCII character dumb terminal is used to connect to the debugger. For serial communications, dBUG requires eight data bits, no parity, one stop bit (8N1), and Xon/Xoff flow control turned off. Xon/Xoff flow control should be turned on when downloading data to the board. The baud rate default is 19200 bps — a speed commonly available from workstations, personal computers and dedicated terminals.

The command line prompt is: dBUG>

Any dBUG command may be entered from this prompt. dBUG does not allow command lines to exceed 80 characters. Wherever possible, dBUG displays data in 80 columns or less. dBUG echoes each character as it is typed, eliminating the need for any local echo on the terminal side.

The <Backspace> and <Delete> keys are recognized as rub-out keys for correcting typographical mistakes.

Command lines may be recalled using the <Control> U, <Control> D and <Control> R key sequences. <Control> U and <Control> D cycle up and down through previous command lines. <Control> R recalls and executes the last command line.

In general, dBUG is not case-sensitive. Commands may be entered either in uppercase or lowercase, depending upon the user's equipment and preference. Only symbol names require that the exact case be used.

Most commands can be recognized by using an abbreviated name. For instance, entering h is the same as entering help. Thus it is not necessary to type the entire command name.

The commands DI, GO, MD, STEP and TRACE are used repeatedly when debugging. dBUG recognizes this and allows for repeated execution of these commands with minimal typing. After a command is entered, press the <Return> or <Enter> key to invoke the command again. The command is executed as if no command line parameters were provided.

2.4 Commands

This section lists the commands that are available with all versions of dBUG. Some board or CPU combinations may use additional commands not listed below.

Table 1. dBUG Command Summary

MNEMONIC	SYNTAX	DESCRIPTION
ASM	asm <<addr> stmt>	Assemble
BC	bc addr1 addr2 length	Block Compare
BF	bf <width> begin end data <inc>	Block Fill
BM	bm begin end dest	Block Move
BR	br addr <-r> <-c count> <-t trigger>	Breakpoint
BS	bs <width> begin end data	Block Search

Motorola Confidential Proprietary

Table 1. dBUG Command Summary (continued)

MNEMONIC	SYNTAX	DESCRIPTION
DC	dc value	Data Convert
DI	di<addr>	Disassemble
DL	dl <offset>	Download Serially
DLDBUG	dldbug	Download dBUG
FL	fl <command> dest <src> size	Erase/Program External Flash
GO	go <addr>	Execute
GT	gt addr	Execute To
HBR	hbr addr <-r>	Hardware Breakpoint
HELP	help <command>	Help
IRD	ird <module.register>	Internal Register Display
IRM	irm module.register data	Internal Register Modify
LR	lr<width> addr	Loop Read
LW	lw<width> addr data	Loop Write
MD	md<width> <begin> <end>	Memory Display
MM	mm<width> addr <data>	Memory Modify
MMAP	mmap	Memory Map Display
RD	rd <reg>	Register Display
RM	rm reg data	Register Modify
RESET	reset	Reset
SD	sd	Stack Dump
SET	set <option value>	Set Configurations
SHOW	show <option>	Show Configurations
STEP	step	Step (Over)
SYMBOL	symbol <symb> <-a symb value> <-r symb> <-C|l|s>	Symbol Management
TRACE	trace <num>	Trace (Into)
VERSION	version	Show Version

ASM Assembler

Usage: ASM <<addr> stmt>

The ASM command is a primitive assembler. The <stmt> is assembled and the resulting code placed at <addr>. This command has an interactive and non-interactive mode of operation.

The value for address <addr> may be an absolute address specified as a hexadecimal value, or a symbol name. The value for stmt must be valid assembler mnemonics for the CPU.

For the interactive mode, the user enters the command and the optional <addr>. If the address is not specified, then the last address is used. The memory contents at the address are disassembled, and the user prompted for the new assembly. If valid, the new assembly is placed into memory, and the address incremented accordingly. If the assembly is not valid, then memory is not modified, and an error message produced. In either case, memory is disassembled and the process repeats.

The user may press the <Enter> or <Return> key to accept the current memory contents and skip to the next instruction, or a enter period to quit the interactive mode.

In the non-interactive mode, the user specifies the address and the assembly statement on the command line. The statement is the assembled, and if valid, placed into memory, otherwise an error message is produced.

Examples:

To place a NOP instruction at address 0xFF01_0000, the command is:

```
asm     FF010000 nop
```

To interactively assembly memory at address 0xFF40_0000, the command is:

```
asm     FF400000
```


Motorola Confidential Proprietary

BC Block Compare

Usage: BC addr1 addr2 length

The BC command compares two contiguous blocks of memory on a byte by byte basis. The first block starts at address addr1 and the second starts at address addr2, both of length bytes.

If the blocks are not identical, the address of the first mismatch is displayed. The value for addresses addr1 and addr2 may be an absolute address specified as a hexadecimal value or a symbol name. The value for length may be a symbol name or a number converted according to the user defined radix (hexadecimal by default).

Example:

To verify that the data starting at 0xFFF2_0000 and ending at 0xFFF3_0000 is identical to the data starting at 0xFFF0_0000, the command is:

```
bc      FFF20000 FFF00000 10000
```

BF Block Fill

Usage: BF<width> begin end data <inc>

The BF command fills a contiguous block of memory starting at address begin, stopping at address end, with the value data. <Width> modifies the size of the data that is written. If no <width> is specified, the default of word sized data is used.

The value for addresses begin and end may be an absolute address specified as a hexadecimal value, or a symbol name. The value for data may be a symbol name, or a number converted according to the user-defined radix, normally hexadecimal.

The optional value <inc> can be used to increment (or decrement) the data value during the fill.

This command first aligns the starting address for the data access size, and then increments the address accordingly during the operation. Thus, for the duration of the operation, this command performs properly-aligned memory accesses.

Examples:

To fill a memory block starting at 0xFF02_0000 and ending at 0xFF04_0000 with the value 0x1234, the command is:

```
bf      FF020000 FF040000 1234
```

To fill a block of memory starting at 0xFFF20000 and ending at 0xFF04_0000 with a byte value of 0xAB, the command is:

```
bf.b    FF020000 FF040000 AB
```

To zero out the BSS section of the target code (defined by the symbols bss_start and bss_end), the command is:

```
bf      bss_start bss_end 0
```

To fill a block of memory starting at 0xFFF2_0000 and ending at 0xFF04_0000 with data that increments by 2 for each <width>, the command is:

```
bf      FF020000 FF040000 0 2
```


Motorola Confidential Proprietary

BM Block Move

Usage: BM begin end dest

The BM command moves a contiguous block of memory starting at address begin and stopping at address end to the new address dest. The BM command copies memory as a series of bytes, and does not alter the original block.

The values for addresses begin, end, and dest may be absolute addresses specified as hexadecimal values, or symbol names. If the destination address overlaps the block defined by begin and end, an error message is produced and the command exits.

Examples:

To copy a block of memory starting at 0xFF04_0000 and ending at 0xFF07_0000 to the location 0xFF00_2000, the command is:

```
bm      FF040000 FF070000 FF002000
```

To copy the target code's data section (defined by the symbols data_start and data_end) to 0xFF00_2000, the command is:

```
bm      data_start data_end FF002000
```

BR Breakpoints

Usage: BR addr <-r> <-c count> <-t trigger>

The BR command inserts or removes software breakpoints at address addr. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name.

If no argument is provided to the BR command, a listing of all defined breakpoints is displayed.

The -r option to the BR command removes a breakpoint defined at address addr. If no address is specified in conjunction with the -r option, then all breakpoints are removed.

Each time a breakpoint is encountered during the execution of target code, its count value is incremented by one. By default, the initial count value for a breakpoint is zero, but the -c option allows setting the initial count for the breakpoint.

Each time a breakpoint is encountered during the execution of target code, the count value is compared against the trigger value. If the count value is equal to or greater than the trigger value, a breakpoint is encountered and control returned to dBUG. By default, the initial trigger value for a breakpoint is one, but the -t option allows setting the initial trigger for the breakpoint.

If no address is specified in conjunction with the -c or -t options, then all breakpoints are initialized to the values specified by the -c or -t option.

Examples:

To set a breakpoint at the C function main() (symbol _main; see "symbol" command), the command is:

```
br      _main
```

To set a breakpoint at 0x003F_A000, the command is:

```
br      3fA000
```

When the target code is executed and the processor reaches 0x003F_8000, control will be returned to dBUG.

To set a breakpoint at the address 0x003F_A000 and set its trigger value to 3, the command is:

```
br      3fA000 -t 3
```

When the target code is executed, the processor must attempt to execute the instruction at 0x003F_A000 a third time before returning control back to dBUG.

To remove all breakpoints, the command is:

```
br      -r
```


Motorola Confidential Proprietary

BS Block Search

Usage: BS<width> begin end data

The BS command searches a contiguous block of memory starting at address begin, stopping at address end, for the value data. <Width> modifies the size of the data that is compared during the search. If no <width> is specified, the default of word sized data is used.

The values for addresses begin and end may be absolute addresses specified as hexadecimal values, or symbol names. The value for data may be a symbol name or a number converted according to the user-defined radix, normally hexadecimal.

This command first aligns the starting address for the data access size, and then increments the address accordingly during the operation. Thus, for the duration of the operation, this command performs properly-aligned memory accesses.

Examples:

To search for the 32-bit value 0x1234_5678 in the memory block starting at 0xFFF4_0000 and ending at 0xFFF7_0000:

```
bs      FFF40000 FFF70000 12345678
```

This reads the 32-bit word located at 0x0004_0000 and compares it against the 32-bit value 0x1234_5678. If no match is found, then the address is incremented to 0x0004_0002 and the next 32-bit value is read and compared.

To search for the 16-bit value 0x1234 in the memory block starting at 0xFFF4_0000 and ending at 0xFFF7_0000:

```
bs      40000 FFF70000 1234
```

This reads the 32-bit word located at 0xFFF4_0000 and compares it against the 16-bit value 0x0000_1234. If no match is found, then the address is incremented to 0xFFF4_0004 and the next 32-bit value is read and compared.

DC — Data Conversion

Usage: DC data

The DC command displays the hexadecimal or decimal value data in hexadecimal, binary, and decimal notation.

The value for data may be a symbol name or an absolute value. If an absolute value passed into the DC command is prefixed by '0x', then data is interpreted as a hexadecimal value. Otherwise data is interpreted as a decimal value.

All values are treated as 32-bit quantities.

Examples:

To display the decimal and binary equivalent of 0x1234, the command is:

```
dc      0x1234
```

To display the hexadecimal and binary equivalent of 1234, the command is:

```
dc      1234
```

DI Disassemble

Usage: DI <addr>

The DI command disassembles target code pointed to by addr. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name.

Wherever possible, the disassembler will use information from the symbol table to produce a more meaningful disassembly. This is especially useful for branch target addresses and subroutine calls.

The DI command attempts to track the address of the last disassembled opcode. If no address is provided to the DI command, then the DI command uses the address of the last opcode that was disassembled.

The DI command is repeatable.

Examples:

To disassemble code that starts at 0xFFF4_0000, the command is:

```
di      FFF40000
```

To disassemble code of the C function main(), the command is:

```
di      _main
```

DL Download Console

Usage: DL <offset>

The DL command performs an S-record download of data obtained from the console, typically through a serial port. The value for offset is converted according to the user-defined radix, normally hexadecimal.

If offset is provided, then the destination address of each S-record is adjusted by offset.

The DL command checks the destination download address for validity. If the destination is an address outside the defined downloadable spaces, then an error message is displayed and downloading aborted.

If the S-record file contains the entry point address, then the program counter is set to reflect this address.

The data should be in the format of a .s19 (S-Record) file. For this board, the downloadable spaces are external flash (0xFF00_0000 - 0xFF00_1FFF for the user's exception table and 0xFF00_2000 - 0xFFEF_FFFF for user code) and internal SRAM (0x003f_A000 - 0x003F_FFFF).

Xon/Xoff flow control needs to be turned on in the terminal window to download data.

Examples:

To download an S-record file through the serial port, the command is:

```
dl
```

To download an S-record file through the serial port, and add an offset to the destination address of 0x40, the command is:

```
dl      0x40
```

DLDBUG Download dBUG

Usage: DLDBUG

The DLDBUG command will download the dBUG monitor to the MPC562BC board. First it will erase all sectors of Flash that dBUG occupies, then it will download the code through the serial port. Upon asking if the user is sure they want to do this, the user should respond by typing “yes” if they want to continue. The DLDBUG command will work at baud rates up to and including 57600.

Xon/Xoff flow control needs to be turned on in the terminal window to download data.

To download the dBUG monitor to the board, the command is:

```
dldbug
```

FL Erase/Program Flash

Usage: FL

FL (e)rase addr bytes

FL (w)rite dest src bytes

The FL command is used to erase the external flash, write to external flash, and display flash device information. Erase and Write operations must be done in sector blocks. dBUG assumes that the user has erased enough memory before writing to it. The destination address must be word (4byte) aligned and the byte count must be in word (4byte) multiples.

Examples:

To view the flash device information, the command is:

```
fl
```

To erase 0x20000 bytes of flash starting at 0xFF000000, the command is:

```
fl erase ff000000 20000
```

To copy 0x40 bytes of data from internal SRAM (0x3FA000) to external flash at 0xFF000000, the command is:

```
fl      write   ff000000 3fa000 0x40
```

GO Execute

Usage: GO <addr>

The GO command executes target code starting at address addr. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name.

If no argument is provided, the GO command begins executing instructions at the current program counter.

When the GO command is executed, all user-defined breakpoints are inserted into the target code, and the context is switched to the target program. Control is only regained when the target code encounters a breakpoint, illegal instruction, or other exception which causes control to be handed back to dBUG.

The GO command is repeatable.

Examples:

To execute code at the current program counter, the command is:

```
go
```

To execute code at the C function main(), the command is:

```
go _main
```

To execute code at the address 0xFF04_0000, the command is:

```
go FF040000
```

GT Execute To

Usage: GT addr

The GT command inserts a temporary software breakpoint at addr and then executes target code starting at the current program counter. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name.This command only works when executing code in SRAM.

When the GT command is executed, all breakpoints are inserted into the target code, and the context is switched to the target program. Control is only regained when the target code encounters a breakpoint, illegal instruction, or other exception which causes control to be handed back to dBUG.

Examples:

To execute code up to the C function bench(), the command is:

```
gt _bench
```

HR Hardware Breakpoints

Usage: HBR addr <-r>

The HBR command inserts or removes hardware breakpoints at address addr. The value for addr may be an absolute address specified as a hexadecimal value. Note that the maximum number of hardware breakpoints allowed is 4.

If no argument is provided to the HBR command, a listing of all defined hardware breakpoints is displayed.

The -r option to the HBR command removes a breakpoint defined at address addr. If no address is specified in conjunction with the -r option, then all hardware breakpoints are removed.

Examples:

To set a breakpoint at the address 0x003F_8000, the command is:

```
hbr     3f8000
```

When the target code is executed and the processor reaches 0x003F_8000, control will be returned to dBUG.

To remove all hardware breakpoints, the command is:

```
hbr     -r
```

HELP Help

Usage: HELP <command>

The HELP command displays a brief syntax of the commands available within dBUG. In addition, the address of where user code may start is given. If command is provided, then a brief listing of the syntax of the specified command is displayed.

Examples:

To obtain a listing of all the commands available within dBUG, the command is:

```
help
```

To obtain help on the breakpoint command, the command is:

```
help br
```

IRD Internal Register Display

Usage: IRD <module.register>

This command displays the internal registers of different modules inside the MPC500. In the command line, module refers to the module name where the register is located and register refers to the specific register to display.

The registers are organized according to the module to which they belong. The available modules on the MPC500 are USIU, TPU_A, TPU_B, QADC_A, QADC_B, QSMCM_A, MIOS14, CAN_A, CAN_B, CAN_C, UIMB, CALRAM_A, CALRAM_B, DPTRAM8K, and PPM. Refer to the MPC562 user's manual for more information on these modules and the registers they contain.

Example:

```
ird     usiu.plprcrk
```

IRM Internal Register Modify

Usage: IRM module.register data

This command modifies the contents of the internal registers of different modules inside the MPC500. In the command line, module refers to the module name where the register is located and register refers to the specific register to modify. The data parameter specifies the new value to be written into the register.

The registers are organized according to the module to which they belong. The available modules on the MPC500 are USIU, TPU_A, TPU_B, QADC_A, QADC_B, QSMCM_A, MIOS14, CAN_A. Refer to the MPC564 user's manual for more information on these modules and the registers they contain.

Example:

To modify the PCPRCR in the USIU to the value 0x0091_4000, the command is:

```
irm     usiu.plprcr 914000
```

LR Loop Read

Usage: LR<width> addr

The LR command continually reads the data at addr until a key is pressed. The optional <width> specifies the size of the data to be read. If no <width> is specified, the command defaults to reading word sized data.

Example:

To continually read the word data from address 0xFFF2_0000, the command is:

```
lr      FFF20000
```

LW Loop Write

Usage: LW<width> addr data

The LW command continually writes data to addr. The optional width specifies the size of the access to memory. The default access size is a word.

Examples:

To continually write the data 0x1234_5678 to address 0xFFF2_0000, the command is:

```
lw      FFF20000 12345678
```

Note that the following command writes 0x78 into memory:

```
lw.b    FFF20000 12345678
```

MD Memory Display

Usage: MD<width> <begin> <end>

The MD command displays a contiguous block of memory starting at address begin and stopping at address end. The values for addresses begin and end may be absolute addresses specified as hexadecimal values, or symbol names. Width modifies the size of the data that is displayed. If no <width> is specified, the default of word sized data is used.

Memory display starts at the address begin. If no beginning address is provided, the MD command uses the last address that was displayed. If no ending address is provided, then MD will display memory up to an address that is 128 beyond the starting address.

This command first aligns the starting address for the data access size, and then increments the address accordingly during the operation. Thus, for the duration of the operation, this command performs properly-aligned memory accesses.

Examples:

To display memory at address 0xFF40_0000, the command is:

```
md FF400000
```

To display memory in the data section (defined by the symbols data_start and data_end), the command is:

```
md data_start
```

To display a range of bytes from 0xFF040000 to 0xFF05_0000, the command is:

```
md.b    FF040000 FF050000
```

To display a range of 32-bit values starting at 0xFF04_0000 and ending at 0xFF05_0000:

```
md      FF040000 FF050000
```

MM Memory Modify

Usage: MM<width> addr <data>

The MM command modifies memory at the address addr. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name. Width specifies the size of the data that is modified. If no <width> is specified, the default of word sized data is used. The value for data may be a symbol name, or a number converted according to the user-defined radix, normally hexadecimal.

For this board, the spaces where MM will work are external flash (0xFF00_0000 - 0xFF00_1FFF for the user's exception table and 0xFF00_2000 - 0xFFEF_FFFF for user code) and internal SRAM (0x003f_A000 - 0x003F_FFFF).Note that if the address to be modified is in external flash, dBUG assumes that the user has erased the necessary block of memory first since flash can only change 1's to 0's.

If a value for data is provided, then the MM command immediately sets the contents of addr to data. If no value for data is provided, then the MM command enters into a loop. The loop obtains a value for data, sets the contents of the current address to data, increments the address according to the data size, and repeats. The loop terminates when an invalid entry for the data value is entered, i.e., a period.

This command first aligns the starting address for the data access size, and then increments the address accordingly during the operation. Thus, for the duration of the operation, this command performs properly-aligned memory accesses.

Examples:

To set the byte at location 0xFF01_0000 to be 0xFF, the command is:

```
mm.b    FF010000 FF
```

To interactively modify memory beginning at 0xFF01_0000, the command is:

```
mm      FF010000
```

MMAP Memory Map Display

Usage: mmap

This command displays the memory map information for the MPC562BC board. The information displayed includes the type of memory, the start and end address of the memory, and the port size of the memory. The display also includes information on how the Chip-selects are used on the board.

Here is an example of the output from this command:

```
Type              Start          End
----------------------------------------
ISB               0x00000000   0x003FFFFF
Internal SRAM     0x003F8000   0x003FFFFF
User SRAM         0x003FA000   0x003FFFFF
Flash             0xFF000000   0xFFFFFFFF
User Flash        0xFF000000   0xFFEFFFFF
dBUG Flash        0xFFF00000   0xFFFFFFFF
```

RD Register Display

Usage: RD <reg>

The RD command displays the register set of the target. If no argument for reg is provided, then all registers are displayed. Otherwise, the value for reg is displayed.

dBUG preserves the registers by storing a copy of the register set in a buffer. The RD command displays register values from the register buffer.

Examples:

To display all the registers and their values, the command is:

```
rd
```

To display only the program counter:

```
rd      pc
```

Here is an example of the output from this command:

```
pc: FFF08000     msr: 00009042 [EE,ME,IP,RI]

cr: 00000000     xer: 00000000      lr: 00000000     ctr: 00000000

r00-07: 00000000 003FA000 00000000 00000000 00000000 00000000 00000000 00000000

r08-15: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

r16-23: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

r24-31: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

RM Register Modify

Usage: RM reg data

The RM command modifies the contents of the register reg to data. The value for reg is the name of the register, and the value for data may be a symbol name, or it is converted according to the user-defined radix, normally hexadecimal.

dBUG preserves the registers by storing a copy of the register set in a buffer. The RM command updates the copy of the register in the buffer. The actual value will not be written to the register until target code is executed.

Examples:

To change program counter to contain the value 0x00ff_2004, the command is:

```
rm      pc ff002004
```

RESET Reset the Board and dBUG

Usage: RESET

The RESET command resets the board and dBUG to their initial power-on states.

The RESET command executes the same sequence of code that occurs at power-on. If the RESET command fails to reset the board adequately, cycle the power or press the reset button.

Examples:

To reset the board and clear the dBUG data structures, the command is:

```
dBUG>   reset
```

SET Set Configurations

Usage: SET <option value>

The SET command allows the setting of user-configurable options within dBUG. With no arguments, SET displays the options and values available. The SHOW command displays the settings in the appropriate format. The standard set of options is listed below.

- baud - This is the baud rate for the serial port on the board. All communications between dBUG and the user occur using either 9600 or 19200 bps, eight data bits, no parity, and one stop bit, 8N1, with no flow control. Xon/Xoff flow control should be turned on when downloading data to the board with the DL or DLDBUG commands.
- base - This is the default radix for use in converting a number from its ASCII text representation to the internal quantity used by dBUG. The default is hexadecimal (base 16), and other choices are binary (base 2), octal (base 8), and decimal (base 10).

Examples:

To set the baud rate of the board to be 57600, the command is:

```
set     baud 57600
```

NOTE: The board must be reset for the baud rate to be changed!

NOTE

See the SHOW command for a display containing the correct formatting of these options.

SHOW Show Configurations

Usage: SHOW <option>

The SHOW command displays the settings of the user-configurable options within dBUG. When no option is provided, SHOW displays all options and values.

Examples:

To display all options and settings, the command is:

```
show
```

To display the current baud rate of the board, the command is:

```
show    baud
```

Here is an example of the output from a show command:

```
dBUG> show

        base: 16

        baud: 19200
```

STEP Step Over

Usage: STEP

The STEP command can be used to "step over" a subroutine call, rather than tracing every instruction in the subroutine. The ST command sets a temporary software breakpoint one instruction beyond the current program counter and then executes the target code. This command only works when executing code in SRAM.

The STEP command can be used to "step over" BSR and JSR instructions.

The STEP command will work for other instructions as well, but note that if the STEP command is used with an instruction that will not return, i.e. BRA, then the temporary breakpoint may never be encountered and dBUG may never regain control.

Examples:

To pass over a subroutine call, the command is:

```
step
```

SYMBOL Symbol Name Management

Usage: SYMBOL <symb> <-a symb value> <-r symb> <-c|l|s>

The SYMBOL command adds or removes symbol names from the symbol table. If only a symbol name is provided to the SYMBOL command, then the symbol table is searched for a match on the symbol name and its information displayed.

The -a option adds a symbol name and its value into the symbol table. The -r option removes a symbol name from the table.

The -c option clears the entire symbol table, the -l option lists the contents of the symbol table, and the -s option displays usage information for the symbol table.

Symbol names contained in the symbol table are truncated to 31 characters. Any symbol table lookups, either by the SYMBOL command or by the disassembler, will only use the first 31 characters. Symbol names are case-sensitive.

Examples:

To define the symbol “main” to have the value 0xFF00_2000, the command is:

```
symbol          -a main FF002000
```

To remove the symbol “junk” from the table, the command is:

```
symbol          -r junk
```

To see how full the symbol table is, the command is:

```
symbol          -s
```

To display the symbol table, the command is:

```
symbol          -l
```

TRACE Trace Into

Usage: TRACE <num>

The TRACE command allows single-instruction execution. If num is provided, then num instructions are executed before control is handed back to dBUG. The value for num is a decimal number.

The TRACE command sets bits in the processors' supervisor registers to achieve single-instruction execution, and the target code executed. Control returns to dBUG after a single-instruction execution of the target code.

This command is repeatable.

Examples:

To trace one instruction at the program counter, the command is:

```
tr
```

To trace 20 instructions from the program counter, the command is:

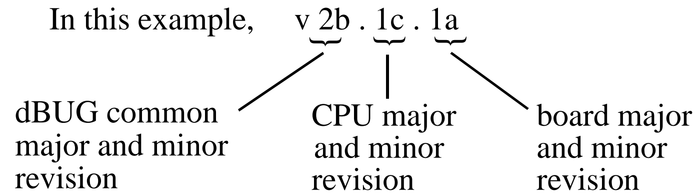
```
tr      20
```

VERSION Display dBUG Version

Usage: VERSION

The VERSION command displays the version information for dBUG. The dBUG version, build number and build date are all given.

The version number is separated by a decimal, for example, “v 2b.1c.1a”.



The version date is the day and time at which the entire dBUG monitor was compiled and built.

Examples:

To display the version of the dBUG monitor, the command is:

```
version
```

2.5 System Call Functions

An additional utility within the dBUG firmware is a function called the System Call handler. This function can be called by the user program to utilize various routines within dBUG, to perform a special task, and to return control to dBUG. This section describes the System Call handler and how it is used.

There are 6 System Call functions. These are: OUT_CHAR, IN_CHAR, IN_STAT, ISR_REGISTER, ISR_REMOVE and EXIT_TO_dBUG. The system call interface accepts an opcode in r10 to indicate which operation is to be performed. Various results are returned, usually in r3. When these routines are invoked, the following is true: sprg0 contains r31 and sprg1 contains LR.

2.5.1 OUT_CHAR

This function (function code 0x0020) sends a character, which is in r3, to terminal. The system call interface accepts an opcode in r10 to indicate which operation is to be performed.

Assembly example:

```
/* assume r3 contains the character */
addi r10, r0, 0x0020        Selects the function
sc                          The character in r3 is sent to terminal
```

C example:

```
/* assume r3 contains the character */
void board_out_char (int ch)
{
        asm(“addi r10, r0, 0x0020”);Selects the function
        asm(“sc”);              The character in r3 is sent to terminal
}
```

2.5.2 IN_CHAR

This function (function code 0x0000) returns an input character (from terminal) to the caller. The returned character is in r3.

Assembly example:

```
/* the character is returned to the user in r3*/
addi r10, r0, 0x0000        Selects the function
sc                          The character is returned in r3
```

C example:

```
int board_in_char (void)
{       /* assume r3 contains the character */
```

```
        asm("addi r10, r0, 0x0000");Selects the function
        asm("sc");          The character is returned in r3
}
```

2.5.3 IN_STAT

This function (function code 0x0001) checks if an input character is present to receive. A value of zero is returned in r3 when no character is present. A value of 1 in r3 means a character is present.

Assembly example:

```
addi r10, r0, 0x0001     Select the function
sc                        Make the call,  r3 contains the response (yes/no).
```

C example:

```
int board_char_present (void)
{
        asm("addi   r10,r0,0x0001");Select the function
        asm("sc");              Make the call,  r3 contains the response (yes/no).
}
```

2.5.4 ISR_REGISTER

This function's code is 0x0040. For ISR_REGISTER, the vector, handler, device ptr, and arg ptr are in r3, r4, r5, and r6 respectively.

C example:

```
int
board_isr_register (int vector, void *handler, void *device, void *arg)
{
        /*
         * Vector will normally be 0x0500 for IRQ.  Handler should be address
         * of your routine.  Device and Arg are both used as arguments to
         * Handler when it is invoked.  Ie. handler(device,arg);  It is
         * intended that Device point to the device , and Arg is meant to point
         * to a data structure to assist the ISR.  If the handler is
         * registered OK, 1 is returned, otherwise 0.
         */
        asm("addi   r10,r0,0x0040");
        asm("sc");
}
```

2.5.5 ISR_REMOVE

This function's code is 0x0041. For ISR_REMOVE, the vector is in r3. Nothing is returned.

Assembly example:

```
addi r10, r0, 0x0041    Selects the function
sc                      The character is returned in r3
```

C example:

```
int
board_isr_remove (void *handler)
{
        asm("addi   r10,r0,0x0041");
        asm("sc");
}
```

2.5.6 EXIT_TO_dBUG

This function (function code 0x0000) transfers the control back to the dBUG, by terminating the user code. The register context is preserved.

Assembly example:

```
lwz      r0,72(r1)
mtspr    8,r0
addi     r1,r1,64
b        asm_sc_exit_to_dbug
```

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

JAPAN:

Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu Minato-ku
Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre, 2 Dai King Street
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong
852-26668334

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

http://www.motorola.com/semiconductors

Information in this document is provided solely to enable system and software implementers to use Motorola products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.

Motorola and the Stylized M Logo are registered in the U.S. Patent and Trademark Office. digital dna is a trademark of Motorola, Inc. All other product or service names are the property of their respective owners. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

© Motorola, Inc. 2002

Using the dBUG
Monitor Firmware