MOTOROLA

dBUG Reference Manual

Revision 0.21

April 8, 1999

PREFACE

The *dBUG Reference Manual* describes the use of dBUG, a ROM monitor/debugger.

This manual is organized as follows:

Section 1: User’s Information
Section 2: Network Operation
Section 3: CPU Specific Information
Section 4: Writing Board Support Packages

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

SECTION 1
USER'S INFORMATION

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.

1.1 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, and one stop bit (8N1). The baud rate may be either 9600 or 19200 bps — speeds 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.

1.2 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.

1.2.1 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 0x00010000, the command is:

```
asm  10000 nop
```

To interactively assembly memory at address 0x00400000, the command is:

```
asm  400000
```

1.2.2 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.

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 0x00010000 and ending at 0x00040000 with the value 0x1234, the command is:

```
bf   10000 40000 1234
```

To fill a block of memory starting at 0x00010000 and ending at 0x0004000 with a byte value of 0xAB, the command is:

```
bf.b 10000 40000 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
```

1.2.3 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 0x00040000 and ending at 0x00080000 to the location 0x00200000, the command is:

```
bm   40000 80000 200000
```

To copy the target code’s data section (defined by the symbols data_start and data_end) to 0x00200000, the command is:

```
bm   data_start data_end 200000
```

1.2.4 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.

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 16-bit value 0x1234 in the memory block starting at 0x00040000 and ending at 0x00080000:

MC68000 and ColdFire:

```
bs   40000 80000 1234
```

PowerPC:

```
bs.h 40000 80000 1234
```

This reads the 16-bit word located at 0x00040000 and compares it against the 16-bit value 0x1234. If no match is found, then the address is incremented to 0x00040002 and the next 16-bit value is read and compared.

To search for the 32-bit value 0xABCD in the memory block starting at 0x00040000 and ending at 0x00080000:

MC68000 and ColdFire:

```
bs.l 40000 80000 ABCD
```

PowerPC:

```
bs   40000 80000 ABCD
```

This reads the 32-bit word located at 0x00040000 and compares it against the 32-bit value 0x0000ABCD. If no match is found, then the address is incremented to 0x00040004 and the next 32-bit value is read and compared.

1.2.5 BR - Breakpoints

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

The BR command inserts or removes breakpoints at address *addr*. The value for *addr* may be an absolute address specified as a hexadecimal value, or a symbol name. *Count* and *trigger* are numbers converted according to the user-defined radix, normally hexadecimal.

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(), the command is:

```
br   _main
```

When the target code is executed and the processor reaches main(), control will be returned to dBUG.

To set a breakpoint at the C function bench() and set its trigger value to 3, the command is:

```
br   _bench -t 3
```

When the target code is executed, the processor must attempt to execute the function bench() a third time before returning control back to dBUG.

To remove all breakpoints, the command is:

```
br   -r
```

1.2.6 DATA - Data Conversion

Usage: DATA *data*

The DATA command displays *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 DATA 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 equivalent of 0x1234, the command is:

```
data  0x1234
```

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

```
data 1234
```

1.2.7 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.

Examples:

To disassemble code that starts at 0x00040000, the command is:

```
di    40000
```

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

```
di   _main
```

1.2.8 DL - Download Console

Usage: DL <*offset*>

The DL command performs an S-record download of data obtained from the console, typically 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 user space, 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.

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 adjust the destination address by 0x40, the command is:

```
dl   0x40
```

1.2.9 DN - Download Network

Usage: DN <-c> <-e> <-i> <-s> <-o *offset*> <*filename*>

The DN command downloads code from the network. The DN command handle files which are either S-record, COFF, ELF or Image formats. The DN command uses Trivial File Transfer Protocol (TFTP) to transfer files from a network host.

In general, the type of file to be downloaded and the name of the file must be specified to the DN command. The -c option indicates a COFF download, the -e option indicates an ELF download, the -i option indicates an Image download, and the -s indicates an S-record download. The -o option works only in conjunction with the -s option to indicate an optional *offset* for S-record download. The *filename* is passed directly to the TFTP server and therefore must be a valid filename on the server.

If neither of the -c, -e, -i, -s or filename options are specified, then a default filename and filetype will be used. Default filename and filetype parameters are manipulated using the SET and SHOW commands.

The DN command checks the destination download address for validity. If the destination is an address outside the defined user space, then an error message is displayed and downloading aborted.

For ELF and COFF files which contain symbolic debug information, the symbol tables are extracted from the file during download and used by dBUG. Only global symbols are kept in dBUG. The dBUG symbol table is not cleared prior to downloading, so it is the user's responsibility to clear the symbol table as necessary prior to downloading.

If an entry point address is specified in the S-record, COFF or ELF file, the program counter is set accordingly.

Examples:

To download an S-record file with the name “srec.out”, the command is:

```
dn -s srec.out
```

To download a COFF file with the name “coff.out”, the command is:

```
dn -c coff.out
```

To download a file using the default filetype with the name “bench.out”, the command is:

```
dn bench.out
```

To download a file using the default filename and filetype, the command is:

```
dn
```

1.2.10 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.

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 0x00040000, the command is:

```
go 40000
```

1.2.11 GT - Execute To

Usage: GT *addr*

The GT command inserts a temporary 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.

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
```

1.2.12 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
```

1.2.13 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.

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 0x00400000, the command is:

```
md 400000
```

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 0x00040000 to 0x00050000, the command is:

```
md.b 40000 50000
```

To display a range of 32-bit values starting at 0x00040000 and ending at 0x00050000:

MC68000 and ColdFire:

```
md.l 40000 50000
```

PowerPC:

```
md.w 40000 50000
```

1.2.14 MM - Memory Modify

Usage: MM*<width> addr <data>*

The MM command modifies memory at the address *addr*. The value for address *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. The value for *data* may be a symbol name, or a number converted according to the user-defined radix, normally hexadecimal.

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 0x00010000 to be 0xFF, the command is:

```
mm.b 10000 FF
```

To interactively modify memory beginning at 0x00010000, the command is:

```
mm   10000
```

1.2.15 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
```

1.2.16 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 register D0 on MC68000 and ColdFire to contain the value 0x1234, the command is:

```
rm   D0 1234
```

To change special-purpose register 8 on PowerPC to contain the value 0x00010000, the command is:

```
rm   spr8 10000
```

1.2.17 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:

```
reset
```

1.2.18 SD - Stack Dump Command

Usage:

The stack dump command allows

1.2.19 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 standard set of options is listed below.

- baud - This is the baud rate for the first 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.
- 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).
- client - This is the network Internet Protocol (IP) address of the board. For network communications, the client IP is required to be set to a unique value, usually assigned by your local network administrator.
- server - This is the network IP address of the machine which contains files accessible via TFTP. Your local network administrator will have this information and can assist in properly configuring a TFTP server if one does not exist.
- gateway - This is the network IP address of the gateway for your local subnetwork. If the client IP address and server IP address are not on the same subnetwork, then this option must be properly set. Your local network administrator will have this information.
- netmask - This is the network address mask to determine if use of a gateway is required. This field must be properly set. Your local network administrator will have this information.
- filename - This is the default filename to be used for network download if no name is provided to the DN command.
- filetype - This is the default file type to be used for network download if no type is provided to the DN command. Valid values are: “srecord”, “coff”, and “elf”.

Different boards or CPUs may utilize additional options not listed above.

Examples:

To set the baud rate of the board to be 19200, the command is:

```
set  baud 19200
```

1.2.20 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
```

1.2.21 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 breakpoint one instruction beyond the current program counter and then executes the target code.

For MC68000 and ColdFire, the STEP command can be used for BSR and JSR instructions.

For PowerPC, the command can be used for BL, BLA, BCL and BCLA 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 on MC68000 and ColdFire or BA on PowerPC, 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
```

1.2.22 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 0x00040000, the command is:

```
symbol    -a main 40000
```

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
```

1.2.23 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.

Examples:

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

```
tr
```

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

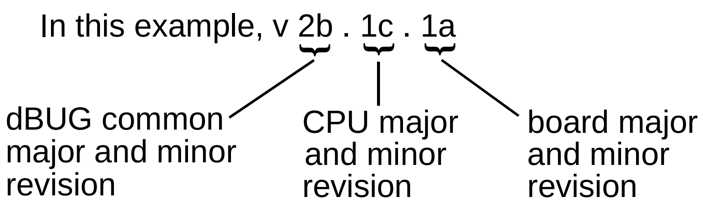
```
tr   20
```

1.2.24 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”.


In this example, v 2b . 1c . 1a
dBUG common major and minor revision
CPU major and minor revision
board major and minor revision

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
```

1.3 DBUG COMMAND SUMMARY

Table 1-1. dBUG Command Summary

ASM	Assemble	ASM <<addr> stmt>
BF	Block Fill	BF<width> begin end data <inc>
BM	Block Move	BM begin end dest
BS	Block Search	BS<width> begin end data
BR	Breakpoint	BR addr <-r> <-c count> <-t trigger>
DATA	Data Convert	DATA value
DI	Disassemble	DI <addr>
DL	Download Serial	DL <offset>
DN	Download Network	DN <-c> <-e> <-i> <-s <-o offset>> <filename>
GO	Execute	GO <addr>
GT	Execute To	GT addr
HELP	Help	help <command>
MD	Memory Display	md<width> <begin> <end>
MM	Memory Modify	mm<width> addr <data>
RD	Register Display	rd <reg>
RM	Register Modify	rm reg data
RESET	Reset	reset
SD	Stack Dump	
SET	Set Configurations	set <option value>
SHOW	Show Configurations	show <option>
STEP	Step (Over)	step
SYMBOL	Symbol Management	symbol <symb> <-a symb value> <-r symb> <-C|l|s>
TRACE	Trace (Into)	trace <num>
VERSION	Show Version	version

SECTION 2
CONFIGURING FOR NETWORK DOWNLOADS

dBUG is capable of downloading over an Ethernet network using the Trivial File Transfer Protocol (TFTP). Prior to using this feature, several parameters are required for network downloads to occur. The information that is required and the steps for configuring dBUG are described in the following paragraphs.

2.1 REQUIRED NETWORK PARAMETERS

For performing network downloads, dBUG needs six parameters; four are network-related, and two are download-related. The parameters are listed below, with the dBUG designation following in parenthesis.

All computers connected to an Ethernet network using the Internet Protocol (IP) need three network-specific parameters. These parameters are:

- IP address for the dBUG-based computer (client)
- IP address of the gateway for non-local traffic (gateway)
- Network IP netmask for flagging traffic as local or non-local (netmask)

In addition, the dBUG network download command requires the following three parameters:

- IP address of the TFTP server (server)
- Default name of the file to download (filename)
- Default type of the file to download (filetype)

Your local system administrator can assign a unique IP address for the board, and also provide you the IP addresses of the gateway, netmask, and TFTP server. Fill out the lines below with this information.

Client:	___.___.___.___	(IP address of the board)
Server:	___.___.___.___	(IP address of the TFTP server)
Gateway:	___.___.___.___	(IP address of the gateway)
Netmask:	___.___.___.___	(Network netmask)

2.2 CONFIGURING DBUG NETWORK PARAMETERS

Once the network parameters have been obtained, dBUG must be configured. The following commands are used to configure the network parameters.

```
set client <client IP>
set server <server IP>
set gateway <gateway IP>
set netmask <netmask>
```

For example, the TFTP server is named ‘santafe’ and has IP address 123.45.67.1. The board is assigned the IP address of 123.45.68.15. The gateway IP address is 123.45.68.250, and the netmask is 255.255.255.0. The commands to dBUG are:

```
set client 123.45.68.15
set server 123.45.67.1
set gateway 123.45.68.250
set netmask 255.255.255.0
```

The last step is to inform dBUG of the name and type of the file to download. Prior to giving the name of the file, keep in mind the following: Most, if not all, TFTP servers will only permit access to files starting at a particular sub-directory. (This is a security feature which prevents reading of arbitrary files by unknown persons.) For example, SunOS uses the directory */tftp_boot* as the default TFTP directory. When specifying a filename to a SunOS TFTP server, all filenames are relative to */tftp_boot*. As a result, you normally will be required to copy the file to download into the directory used by the TFTP server.

A default filename for network downloads is maintained by dBUG. To change the default filename, use the command:

```
set filename <filename>
```

When using the Ethernet network for downloading, either S-record, COFF, Elf, or Image files may be downloaded. A default filetype for network downloads is maintained by dBUG as well. To change the default filetype, use the command:

```
set filetype <srecord|coff|elf|image>
```

Continuing with the above example, the compiler produces an executable COFF file, *a.out*. This file is copied to the */tftp_boot* directory on the server with the command:

```
rcp a.out santafe:/tftp_boot/a.out
```

Change the default filename and filetype with the commands:

```
set filename a.out
set filetype coff
```

Finally, perform the network download with the DN command. The network download process uses the configured IP addresses and the default filename and file type for initiating a TFTP download from the TFTP server.

2.3 TROUBLESHOOTING NETWORK PROBLEMS

Most problems related to network downloads are a direct result of improper configuration. Verify that all IP addresses configured into dBUG are correct. This is accomplished via the SHOW command.

Using an IP address that is already assigned to another machine will cause the dBUG network download to fail, and will probably cause other severe network problems. Make certain the client IP address is unique for the board.

Check for proper insertion or connection of the network cable. Are status LEDs lit to indicate that network traffic is present?

Check for proper configuration and operation of the TFTP server. Most Unix workstations can execute the command TFTP which can be used to connect to the TFTP server as well. Is the default TFTP root directory present and readable?

If `ICMP_DESTINATION_UNREACHABLE` or similar ICMP messages appear, then a serious error has occurred. Reset the board and wait one minute for the TFTP server to time out and terminate any open connections. Verify that the IP addresses for the server and gateway are correct.

SECTION 3
CPU-SPECIFIC INFORMATION

This section provides information concerning the use and management of the CPU resources by dBUG.

3.1 M68000 FAMILY

For the M68000 family of processors, the following generalizations are true.

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

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no *<width>* option is provided, the default width is .W, 16 bits.

The core MC68000 register set is maintained by dBUG. These are listed below:

- A0-A7
- D0-D7
- PC
- SR

dBUG supports the following processors:

- MC68EC020
- MC68EC030
- MC68EC040

More complete descriptions of these CPUs are given in the following paragraphs.

3.1.1 MC68EC020

The MC68EC020 is an embedded controller with an on-chip instruction cache for faster instruction executions. A 24-bit address bus and 32-bit data bus provide dynamic bus sizing for simple bus access to 8-, 16-, and 32-bit devices. The MC68EC020 provides full support of virtual memory and virtual machine. A new bit field data type accelerates bit-oriented applications — e.g. video graphics.

3.1.1.1 COMMAND LINE OPTIONS. For commands that accept an optional *<width>* to modify the memory access size, the valid values are:

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no *<width>* option is provided, the default width is .W, 16 bits.

3.1.1.2 CPU-SPECIFIC COMMANDS. None.

3.1.1.3 REGISTER SET. dBUG maintains these registers.

- A0-A7 (A7 is USP)
- D0-D7
- PC
- SR
- ISP, MSP, VBR, SFC, DFC, CACR, CAAR, AC0, AC1, ACUSR

3.1.1.4 EXCEPTION PROCESSING. The following actions are performed when an exception occurs.

- The register context is stored in the register context buffer.
- Caches are flushed and disabled.
- Appropriate processing of the exception.

If code execution is to continue after exception handling, the following actions are taken.

- If caches were previously enabled, then caches are turned on again.
- The register context is restored.
- An RTE instruction starts execution of the code again.

The following actions are performed when an interrupt occurs.

- The volatile registers (as per the ABI) are stored on the stack.
- The appropriate interrupt handler is invoked.

The return value dictates whether execution continues or the register context stored and control returned to dBUG.

3.1.1.5 BOARD SUPPORT PACKAGES. Out of reset, the code in file *mc68ec020_lo.s* executes to perform basic initialization of the MC68EC020 internal resources.

1. A jump to label asm_llinit performs low-level board-specific initialization (i.e., memory controller). When complete, the low-level board-specific initialization code jumps back to label to asm_lldone.
2. The VBR register is written with the address of the dBUG vector table.
3. Both CACR and CAAR registers are written with zeros to disable caching. AC0, AC1 and ACUSR are not touched.

Consult the board support package code for additional information on the initialization sequence of the MC68EC020.

3.1.1.6 MISCELLANEOUS NOTES. The MD and MM commands do not honor the values of the SFC and DFC registers.

For MC68000 documentation, order Motorola Literature part MC68000PM/AD.

For MC68EC020 documentation, order Motorola Literature part MC68020UM/AD.

3.1.2 MC68EC030

The MC68EC030 is a 32-bit embedded controller with two access control registers that define cacheable blocks for the independent 256-byte data and instruction caches. Dynamic bus sizing provides simple bus access to 8-, 16-, and 32-bit devices, and burst mode allows efficient DRAM interface.

3.1.2.1 COMMAND LINE OPTIONS. For commands that accept an optional *<width>* to modify the memory access size, the valid values are:

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no *<width>* option is provided, the default width is .W, 16 bits.

3.1.2.2 CPU-SPECIFIC COMMANDS. None.

3.1.2.3 REGISTER SET. dBUG maintains these registers.

- A0-A7 (A7 is USP)
- D0-D7
- PC
- SR
- ISP, MSP, VBR, SFC, DFC, CACR, CAAR, AC0, AC1, ACUSR

3.1.2.4 EXCEPTION PROCESSING. The following actions are performed when an exception occurs.

- The register context is stored in the register context buffer.
- Caches are flushed and disabled.
- Appropriate processing of the exception.

If code execution is to continue after exception handling, the following actions are taken.

- If caches were previously enabled, then caches are turned on again.
- The register context is restored.
- An RTE instruction starts execution of the code again.

The following actions are performed when an interrupt occurs.

- The volatile registers (as per the ABI) are stored on the stack.
- The appropriate interrupt handler is invoked.

The return value dictates whether execution continues or the register context stored and control returned to dBUG.

3.1.2.5 BOARD SUPPORT PACKAGES. Out of reset, the code in file *mc68ec030_lo.s* executes to perform basic initialization of the MC68EC030 internal resources.

1. A jump to label asm_llinit performs low-level board-specific initialization (i.e. memory controller). When complete, the low-level board-specific initialization code jumps back to label asm_lldone.
2. The VBR register is written with the address of the dBUG vector table.
3. Both CACR and CAAR registers are written with zeros to disable caching.
4. AC0, AC1 and ACUSR registers are written with zeros.

Consult the board support package code for additional information on the initialization sequence of the MC68EC030.

3.1.2.6 MISCELLANEOUS NOTES. The MD and MM commands do not honor the values of the SFC and DFC registers.

For MC68000 documentation, order Motorola Literature part MC68000PM/AD.

For MC68EC0320 documentation, order Motorola Literature part MC68030UM/AD.

3.1.3 MC68EC040

The MC68EC040 is a 32-bit embedded controller that contains fully independent 2 kbyte instruction and data caches, and retains the high-performance integer unit and execution parallelism of the MC68040 microprocessor.

3.1.3.1 COMMAND LINE OPTIONS. For commands that accept an optional *<width>* to modify the memory access size, the valid values are:

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no *<width>* option is provided, the default width is .W, 16 bits.

3.1.3.2 CPU-SPECIFIC COMMANDS. None.

3.1.3.3 REGISTER SET. dBUG maintains these registers.

- A0-A7 (A7 is USP)
- D0-D7
- PC
- SR
- ISP, MSP, VBR, SFC, DFC, CACR, DACR0, DACR1, IACR0, IACR1

3.1.3.4 EXCEPTION PROCESSING. The following actions are performed when an exception occurs.

- The register context is stored in the register context buffer.
- Caches are flushed and disabled.
- Appropriate processing of the exception.

If code execution is to continue after exception handling, the following actions are taken.

- If caches were previously enabled, then caches are turned on again.
- The register context is restored.
- An RTE instruction starts execution of the code again.

The following actions are performed when an interrupt occurs.

- The volatile registers (as per the ABI) are stored on the stack.
- The appropriate interrupt handler is invoked.

The return value dictates whether execution continues or the register context stored and control returned to dBUG.

3.1.3.5 BOARD SUPPORT PACKAGES. Out of reset, the code in file *mc68ec040_lo.s* executes to perform basic initialization of the MC68EC040 internal resources.

1. A jump to label asm_llinit performs low-level board-specific initialization (i.e. memory controller). When complete, the low-level board-specific initialization code jumps back to label asm_lldone.
2. The VBR register is written with the address of the dBUG vector table.
3. The CACR, DACR0, DACR1, IACR0 and IACR1 registers are written with zeros to disable caching.
4. Both the instruction and data caches are invalidated.

Consult the board support package code for additional information on the initialization sequence of the MC68EC040.

3.1.3.6 MISCELLANEOUS NOTES. The MD and MM commands do not honor the values of the SFC and DFC registers.

For MC68000 documentation, order Motorola Literature part MC68000PM/AD.

For MC68EC040 documentation, order Motorola Literature part MC68040UM/AD.

3.2 COLDFIRE MCF5200

For the ColdFire MCF5200 family of processors, the following generalizations are true.

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

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no *<width>* option is provided, the default width is .W, 16 bits.

The core MCF5200 register set is maintained by dBUG. These are listed below:

- A0-A7
- D0-D7
- PC
- SR

dBUG supports the following processors:

- MCF5200/D
- MCF5202
- MCF5204
- MCF5206
- MCF5206e
- MCF5307

More complete descriptions of these CPUs are given in the following paragraphs.

3.2.1 MCF5200/D

The MCF5200/D processor is a developers' chip which contains only the ColdFire core.

3.2.1.1 COMMAND LINE OPTIONS. For commands that accept an optional *<width>* to modify the memory access size, the valid values are:

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no *<width>* option is provided, the default width is .W, 16 bits.

3.2.1.2 CPU-SPECIFIC COMMANDS. None.

3.2.1.3 REGISTER SET. dBUG maintains these registers.

- A0-A7
- D0-D7
- PC
- SR

None of the ColdFire control registers are readable by the supervisor programming model, so they are not accessible via dBUG. User code may change these registers, but caution must be exercised as changes may render dBUG useless.

3.2.1.4 EXCEPTION PROCESSING. When an exception occurs, the registers context is stored in the register context buffer and the exception is handled appropriately. If execution is to continue after the exception, then the register context is restored and an RTE instruction starts code execution again.

When an interrupt occurs, the volatile registers (as per the ABI) are stored on the stack and the appropriate interrupt handler invoked. The return value dictates whether execution continues or the register context stored and control returned to dBUG.

3.2.1.5 BOARD SUPPORT PACKAGES. Out of reset, the code in file *mcf5200_lo.s* executes to perform basic initialization of the MCF5200 internal resources.

1. A jump to label asm_llinit performs low-level board-specific initialization (i.e., memory controller). When complete, the low-level board-specific initialization code jumps back to label asm_lldone.
2. The VBR register is written with the address of the dBUG vector table.

Consult the board support package code for additional information on the initialization sequence of the MCF5200.

3.2.1.6 MISCELLANEOUS NOTES. Errata: MOVEC to VBR actually loads the VBR with the contents of A7.

The MCF5200/D version of dBUG can support all MCF5200 ColdFire processors.

For ColdFire documentation, order Motorola Literature part MCF5200PRM/AD.

3.2.2 MCF5202

The MCF5202 contains the ColdFire core, along with 2K of unified instruction and data cache.

3.2.2.1 COMMAND LINE OPTIONS. For commands that accept an optional *<width>* to modify the memory access size, the valid values are:

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no *<width>* option is provided, the default width is .W, 16 bits.

3.2.2.2 CPU-SPECIFIC COMMANDS. None.

3.2.2.3 REGISTER SET. dBUG maintains these registers.

- A0-A7
- D0-D7
- PC
- SR

None of the ColdFire control registers are readable by the supervisor programming model, so they are not accessible via dBUG. User code may change these registers, but caution must be exercised as changes may render dBUG useless.

3.2.2.4 EXCEPTION PROCESSING. The following actions are performed when an exception occurs.

- The register context is stored in the register context buffer.
- The unified cache is flushed and disabled.
- Appropriate processing of the exception.

If code execution is to continue after exception handling, the following actions are taken.

- The register context is restored.
- An RTE instruction starts execution of the code again.

The following actions are performed when an interrupt occurs.

- The volatile registers (as per the ABI) are stored on the stack.
- The appropriate interrupt handler is invoked.

The return value dictates whether execution continues or the register context stored and control returned to dBUG.

3.2.2.5 BOARD SUPPORT PACKAGES. Out of reset, the code in file *mcf5202_lo.s* executes to perform basic initialization of the MCF5202 internal resources.

1. A jump to label asm_llinit performs low-level board-specific initialization (i.e. memory controller). When complete, the low-level board-specific initialization code jumps back to label asm_lldone.
2. The VBR register is written with the address of the dBUG vector table.
3. The CACR register is written to invalidate and disable the unified cache.
4. The ACR0 and ACR1 registers are written with zeros to disable them.

Consult the board support package code for additional information on the initialization sequence of the MCF5202.

3.2.2.6 MISCELLANEOUS NOTES. Errata: MOVEC to VBR actually loads the VBR with the contents of A7.

For ColdFire documentation, order Motorola Literature part MCF5200PRM/AD.

For MCF5202 documentation, order Motorola Literature part MCF5202UM/AD.

3.2.3 MCF5204

The MCF5204 integrates a ColdFire core with on-chip SRAM, UART, timers and chip selects. The MCF5204 includes a 512-byte instruction cache.

3.2.3.1 COMMAND LINE OPTIONS. For commands that accept an optional *<width>* to modify the memory access size, the valid values are:

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no *<width>* option is provided, the default width is .W, 16 bits.

3.2.3.2 CPU-SPECIFIC COMMANDS. None.

3.2.3.3 REGISTER SET. dBUG maintains these registers.

- A0-A7
- D0-D7
- PC
- SR

None of the ColdFire control registers are readable by the supervisor programming model, so they are not accessible via dBUG. User code may change these registers, but caution must be exercised as changes may render dBUG useless.

3.2.3.4 EXCEPTION PROCESSING. The following actions are performed when an exception occurs.

- The register context is stored in the register context buffer.
- The instruction cache is invalidated and disabled.
- Appropriate processing of the exception.

If code execution is to continue after exception handling, the following actions are taken.

- The register context is restored.
- An RTE instruction starts execution of the code again.

The following actions are performed when an interrupt occurs.

- The volatile registers (as per the ABI) are stored on the stack.
- The appropriate interrupt handler is invoked.

The return value dictates whether execution continues or the register context stored and control returned to dBUG.

3.2.3.5 BOARD SUPPORT PACKAGES. Out of reset, the code in file *mcf5204_lo.s* executes to perform basic initialization of the MCF5204 internal resources.

1. A jump to label asm_llinit performs low-level board-specific initialization (i.e. external memory controller). When complete, the low-level board-specific initialization code jumps back to label asm_lldone.
2. The VBR register is written with the address of the dBUG vector table.
3. The CACR register is written to invalidate and disable the instruction cache.
4. The ACR0 and ACR1 registers are written with zeros to disable them.
5. The RAMBAR is temporarily mapped to provide a working stack space for the initialization code.
6. The function mcf5204_mbar() is called to obtain the address at which to map the MBAR.
7. The function mcf5204_rambar() is called to obtain the address at which to map the RAMBAR.
8. The MBAR and RAMBAR are mapped to the appropriate addresses.
9. The function mcf5204_init() is called to perform all remaining initialization of the MCF5204 internal peripherals.
10. The stack pointer is changed to point to RAM rather than internal SRAM.

Consult the board support package code for additional information on the initialization sequence of the MCF5204.

3.2.3.6 MISCELLANEOUS NOTES. Errata: MOVEC to VBR actually loads the VBR with the contents of A7.

For ColdFire documentation, order Motorola Literature part MCF5200PRM/AD.

For MCF5204 documentation, order Motorola Literature part MCF5204UM/AD.

3.2.4 MCF5206

The MCF5206 and MCF5206e integrate a ColdFire core with on-chip SRAM, two UARTs, timers, chip selects and a DRAM controller. They also include 512 byte instruction cache.

3.2.4.1 COMMAND LINE OPTIONS. For commands that accept an optional *<width>* to modify the memory access size, the valid values are:

.B	8-bit (byte) access
.W	16-bit (word) access
.L	32-bit (long) access

When no *<width>* option is provided, the default width is .W, 16 bits.

3.2.4.2 CPU-SPECIFIC COMMANDS. None.

3.2.4.3 REGISTER SET. dBUG maintains these registers.

- A0-A7
- D0-D7
- PC
- SR

None of the ColdFire control registers are readable by the supervisor programming model, so they are not accessible via dBUG. User code may change these registers, but caution must be exercised as changes may render dBUG useless.

3.2.4.4 EXCEPTION PROCESSING. The following actions are performed when an exception occurs.

- The register context is stored in the register context buffer.
- The instruction cache is invalidated and disabled.
- Appropriate processing of the exception.

If code execution is to continue after exception handling, the following actions are taken.

- The register context is restored.
- An RTE instruction starts execution of the code again.

The following actions are performed when an interrupt occurs.

- The volatile registers (as per the ABI) are stored on the stack.
- The appropriate interrupt handler is invoked.

The return value dictates whether execution continues or the register context stored and control returned to dBUG.

3.2.4.5 BOARD SUPPORT PACKAGES. Out of reset, the code in file *mcf5206_lo.s* executes to perform basic initialization of the MCF5206 internal resources.

1. A jump to label asm_llinit performs low-level board-specific initialization (i.e. external memory controller). When complete, the low-level board-specific initialization code jumps back to label asm_lldone.
2. The VBR register is written with the address of the dBUG vector table.
3. The CACR register is written to invalidate and disable the instruction cache.
4. The ACR0 and ACR1 registers are written with zeros to disable them.
5. The RAMBAR is temporarily mapped to provide a working stack space for the initialization code.
6. The function mcf5206_mbar() is called to obtain the address at which to map the MBAR.
7. The function mcf5206_rambar() is called to obtain the address at which to map the RAMBAR.
8. The MBAR and RAMBAR are mapped to the appropriate addresses.
9. The function mcf5206_init() is called to perform all remaining initialization of the MCF5206 internal peripherals.
10. The stack pointer is changed to point to RAM rather than internal SRAM.

Consult the board support package code for additional information on the initialization sequence of the MCF5206.

3.2.4.6 MISCELLANEOUS NOTES. Errata: MOVEC to VBR actually loads the VBR with the contents of A7.

For ColdFire documentation, order Motorola Literature part MCF5200PRM/AD.

For MCF5206 documentation, order Motorola Literature part MCF5206UM/AD.

3.3 POWERPC

For the PowerPC family of processors, the following generalizations are true.

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 bits.

The core PowerPC register set is maintained by dBUG.

- GPR0-31 (referenced as R0-R31)
- IP (SRR0 is IP)
- MSR (SRR1 is MSR)
- CR, CTR, XER, LR

Most registers are accessible via their symbolic names as well as the special-purpose register number. For instance, the Link Register, SPR8, can be referenced as both “spr8” and “LR”.

dBUG supports the following processors:

- MPC6XX
- MPC8XX

More complete descriptions of these CPUs are provided in the following sections.

3.3.1 MPC6XX

The MPC6XX family of processors contain a high performance PowerPC core and large on-chip caches. The MPC6XX version of dBUG supports these PowerPC processors:

- MPC602
- MPC603, MPC603e, MPC603ev
- MPC604, MPC604e, MPC604ev
- MPC740, MPC750

3.3.1.1 COMMAND LINE OPTIONS. 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 bits.

3.3.1.2 CPU-SPECIFIC COMMANDS. The IBR command is used to enable a hardware instruction breakpoint.

3.3.1.3 REGISTER SET. dBUG maintains these registers.

- GPR0-31 (referenced as r0-r31)
- FPSCR, FPR0-31 (referenced as f0-f31)
- IP (SRR0 is IP)
- MSR (SRR1 is MSR)
- CR, CTR, XER, LR
- DEC, PVR, TBL, TBU and SR0-15
- IBATxL, IBATxU, DBATxL, DBATxU, SDR1, DAR, and DSISR

Additional registers are maintained according to the MPC6XX processor in the system.

MPC602:

None.

MPC603:

- HID0, DMISS, DCMP, HASH1, HASH2, IMISS, ICMP and RPA
- IABR and EAR

MPC603e and MPC603ev:

- HID0, HID1, DMISS, DCMP, HASH1, HASH2, IMISS, ICMP and RPA
- IABR and EAR

MPC604:

- HID0, PMC1, PMC2, MMCR0, SDA, SIA, IABR, DABR, EAR and PIR

MPC604e and MPC604ev:

- HID0, PMC1, PMC2, PMC3, PMC4, MMCR0 and MMCR1
- SDA, SIA, IABR, DABR, EAR and PIR

MPC740 and MPC750:

- UPMC1, UPMC2, UPMC3, UPMC4, USIA, UMMCR0, UMMCR1, HID0 and HID1
- PMC1, PMC2, PMC3, PMC4, MMCR0, MMCR1 and SIA
- THRM1, THRM2, THRM3, ICTC, L2CR, IABR, DABR and EAR

Most registers are accessible via their symbolic names as well as the special-purpose register number. For instance, the Link Register, SPR8, can be referenced as both “spr8” and “LR”.

3.3.1.4 EXCEPTION PROCESSING. The following actions are performed when an exception occurs.

- The register context is stored in the register context buffer.
- The processor caches are flushed/invalidated and disabled.
- The MPC10X L2 caches are flushed and disabled.
- Appropriate processing of the exception.

If code execution is to continue after exception handling, the following actions are taken.

- The register context is restored.
- An RFI instruction starts execution of the code again.

The following actions are performed when an interrupt occurs.

- The volatile registers (as per the ABI) are stored on the stack.
- The appropriate interrupt handler is invoked.

The return value dictates whether execution continues or the register context stored and control returned to dBUG.

3.3.1.5 BOARD SUPPORT PACKAGES. Out of reset, the code in file *mpc6xx_lo.s* executes to perform basic initialization of the MPC6XX internal resources.

1. The MSR is written to place the processor in supervisor and big-endian modes with

instruction and data translation disabled.

2. The HID0 is written to invalidate the caches.
3. The TLBs are invalidated.
4. The floating-point registers are zeroed.
5. All special-purpose registers are zeroed to disable them.
6. A subroutine call to label mpc10x_init performs low-level board-specific initialization of the external memory controller.
7. The stack pointer is changed to point into RAM.

Consult the board support package code for additional information on the initialization sequence of the MPC6XX.

3.3.1.6 MISCELLANEOUS NOTES. Groups of special-purpose, floating-point and segment registers can be displayed with the RD command.

```
rd sprs
rd fprs
rd srs
```

For PowerPC documentation, order Motorola Literature part MPCFPE/AD.

For MPC602 documentation, order Motorola Literature part MPC602/AD.

For MPC603 documentation, order Motorola Literature part MPC603/AD.

For MPC603e documentation, order Motorola Literature part MPC603E/AD.

For MPC604 documentation, order Motorola Literature part MPC604/AD.

For MPC604e documentation, order Motorola Literature part MPC604E/AD.

For MPC750 documentation, order Motorola Literature part MPC750/AD.

3.3.2 MPC8XX

The MPC8XX family of processors integrate a PowerPC core with on-chip caches, a very flexible memory controller and a communications module. The MPC8XX version of dBUG supports these PowerPC processors:

- MPC821
- MPC823
- MPC850
- MPC860, MPC860SAR, MPC860T

3.3.2.1 COMMAND LINE OPTIONS. 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 bits.

3.3.2.2 CPU-SPECIFIC COMMANDS. The IRD command displays the contents of the internal peripheral registers.

The IMM command modifies the contents of the internal peripheral registers.

3.3.2.3 REGISTER SET. dBUG maintains these registers.

- GPR0-31 (referenced as r0-r31)
- IP (SRR0 is IP)
- MSR (SRR1 is MSR)
- CR, CTR, XER, LR, DEC, PVR, TBL, TBU, DSISR, DAR and IMMR
- IC_CST, IC_ADR, IC_DAT, DC_CST, DC_ADR and DC_DAT
- MI_CTR, MI_AP, MI_EPN, MI_TWC and MI_RPN
- MI_DBCAM, MI_DBRAM0 and MI_DBRAM1
- MD_CTR, MD_CASID, MD_AP, MD_EPN, MD_TWC and MD_RPN
- M_TW, MD_DBCAM, MD_DBRAM0 and MD_DBRAM1

Most registers are accessible via their symbolic names as well as the special-purpose register number. For instance, the Link Register, SPR8, can be referenced as both “spr8” and “LR”.

3.3.2.4 EXCEPTION PROCESSING. The following actions are performed when an exception occurs.

- The register context is stored in the register context buffer.
- The processor caches are flushed/invalidated and disabled.
- Appropriate processing of the exception.

If code execution is to continue after exception handling, the following actions are taken.

- The register context is restored.
- An RFI instruction starts execution of the code again.

The following actions are performed when an interrupt occurs.

- The volatile registers (as per the ABI) are stored on the stack.
- The appropriate interrupt handler is invoked.

The return value dictates whether execution continues or the register context stored and control returned to dBUG.

3.3.2.5 BOARD SUPPORT PACKAGES. Out of reset, the code in file *mpc8xx_lo.s* executes to perform basic initialization of the MPC8XX internal resources.

1. The MSR is written to place the processor in supervisor and big-endian modes with instruction and data translation disabled.
2. Both instruction and data caches are invalidated.
3. The TLBs are invalidated.
4. A temporary stack is created in the internal dual-ported RAM.
5. Function mpc8xx_isb() is called to obtain the address at which to map the internal peripherals.
6. The internal peripherals are mapped to the appropriate address.
7. Function mpc8xx_init() is called to perform all initialization of the MPC8XX resources.
8. The stack pointer is changed to point into RAM.

Consult the board support package code for additional information on the initialization sequence of the MPC8XX.

3.3.2.6 MISCELLANEOUS NOTES. The special-purpose registers can be displayed with the RD command.

```
rd sprs
```

For PowerPC documentation, order Motorola Literature part MPCFPE/AD.

For MPC821 documentation, order Motorola Literature part MPC821UM/AD.

For MPC823 documentation, order Motorola Literature part MPC823UM/AD.

For MPC850 documentation, order Motorola Literature part MPC850UM/AD.

For MPC860 documentation, order Motorola Literature part MPC860UM/AD.

SECTION 4
WRITING DBUG BOARD SUPPORT PACKAGES

dBUG is a traditional ROM monitor/debugger designed for rapid platform bring-up. By providing a small number of well-defined functions, dBUG can be fitted to a new hardware platform very quickly.

4.1 OVERVIEW

Written in the C language, dBUG is a portable debugger engineered for systems designed around Motorola’s MC68000, ColdFire and PowerPC processor architectures. dBUG provides a common debugging interface for all these hardware systems. To accomplish this, dBUG is modularized into three components:

- User interface component
- CPU-specific component
- Board-specific component

The user interface component consists of a set of standard commands which provide basic debugging facilities. These commands are the same on all systems.

The CPU-specific component implements all details and services specific to the processor. The user interface and board-specific components draw upon resources provided by the CPU-specific component.

The board-specific component implements all the remaining services required by the user interface and CPU-specific components. These services include platform initialization and basic character input and output. The board-specific component also implements additional commands and features which are required by the particular system.

The steps needed to write a board support package (BSP) for dBUG are detailed in the sections below. Each BSP is called a project, in dBUG terminology. When appropriate, examples are provided based upon the generic project template provided with dBUG.

4.2 DIRECTORY STRUCTURE

The first step in writing a board support package is understanding the development environment provided by dBUG. The dBUG source tree depicted in Figure 4-1 reflects its modularity.

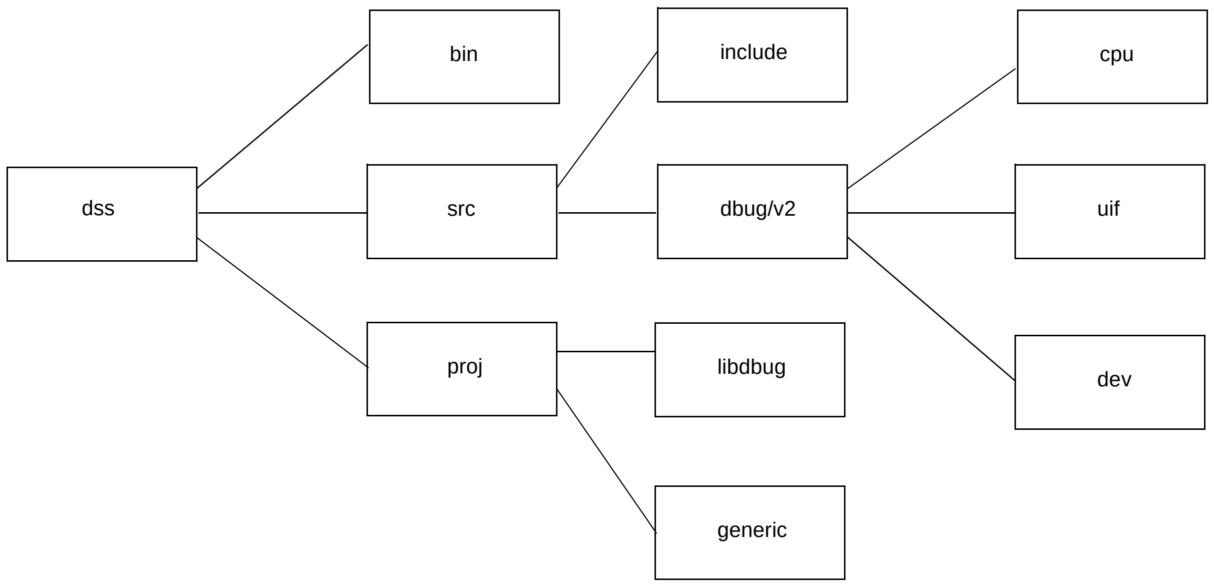


Figure 4-1. dBUG Source Tree

The directory *dss/bin* contains miscellaneous tools for building projects.

The directory *dss/src/dbug/v2/uif* contains the source to the user interface component.

The directory *dss/src/dbug/v2/cpu* contains the source to the CPU-specific component. An entire subdirectory structure and source code base exists for the various processors supported by dBUG.

The directory *dss/src/dbug/v2/dev* contains sample polling drivers for character input and output.

The directory *dss/src/include* contains C header files for dBUG as well as processor header files.

The directory *dss/proj* is where the source for the various BSPs is located. Under this directory, each project has its own subdirectory. Projects must be located here, as all paths to source files are relative to the project subdirectory. Figure 4-2 depicts the subdirectory structure under each project.

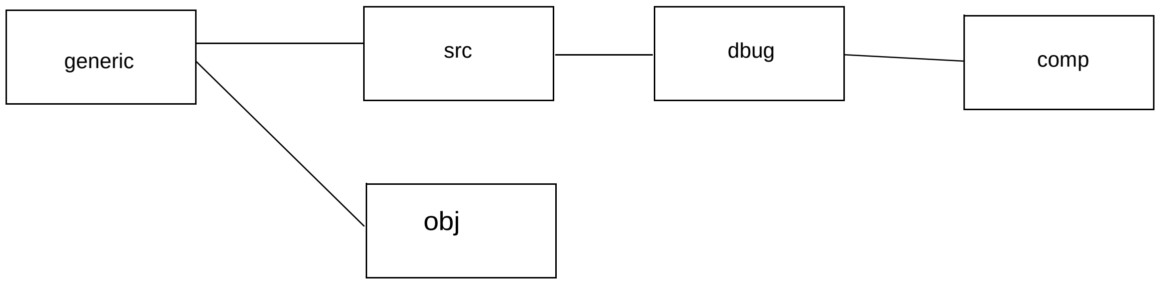


Figure 4-2. Project Directory Structure

The directory *generic/src/dbug* contains the board-specific files. These files and the services they provide are discussed in detail in the subsequent paragraphs.

The directory *generic/src/dbug/comp* contains the host toolchain specific files. Typically these files are makefiles, linker script files and any assembly source files. The directory *comp* will reflect the toolchain in use; for example, gnu or diab.

The directory *generic/obj* contains all object and listing files. In addition, the final dBUG image is placed here when the build process completes.

Located in *dss/proj/generic* is the top-level makefile for building dBUG. The top-level makefile invokes a subordinate makefile which performs the actual work.

The top-level and subordinate makefile scheme allows a single project source code base to be compiled for various target CPUs and toolchains by invoking the appropriate subordinate makefile. For example, the Motorola Integrated Development Platform supports MC680X0, MC683XX and MCF5202 processors. With the appropriate subordinate makefiles, the same BSP source files build dBUG for each of the supported processor and toolchain combinations.

4.3 FILES IN THE BOARD SUPPORT PACKAGE

A number of files are needed to complete a board support package. Source files are needed for the BSP itself, as well as makefiles and linker script files. The following files typically exist for a dBUG project.

- *config.h*
- *board.h*
- *board.c*
- *cmds.c*
- *comp/make.cpu*
- *comp/board.lnk*
- *comp/libdbug.a*
- *comp/vectors.o*

- *comp/scif.s*
- *comp/sc.c*

File *config.h* is required to contain one item: the type of processor. A #define identifies the CPU in use, which in turn directly affects which header files are automatically included. This file may optionally #define DBUG_NETWORK to indicate that the TFTP network download capability is to be available.

File *board.h* contains configuration information, definitions and prototypes specific to the platform. Normally this file is #included by *config.h* to make it visible to all files in the BSP.

File *board.c* contains the board-specific routines that are required by dBUG. These routines are listed in Table 4-1. Additional information is provided in 4.9 Board-Specific Functions.

Table 4-1. Required Board-Specific Functions

FUNCTION	DESCRIPTION
board_init()	Board initialization function
board_init2()	Board initialization function
board_init3()	Board initialization function
board_getchar()	Character input
board_putchar()	Character output
board_getchar_present()	Test for character input
board_putchar_flush()	Flush character output
board_dlio_init()	Download Initialization
board_dlio_getchar()	Download character input
board_dlio_vda()	Download valid address
board_dlio_done()	Download completion
board_get_baud()	Get baud rate of dBUG port
board_set_baud()	Set baud rate for dBUG port
board_reset()	Board reset

File *cmds.c* contains the dBUG command set as well as the SET/SHOW options.

Other board-specific files for drivers, diagnostics or commands are located here as well.

The toolchain specific files are isolated into the *comp* subdirectory. Typically the files located here are the subordinate makefile (*comp/make.cpu*), linker script (*comp/board.lnk*), board-specific system calls (*comp/scif.s* and *comp/sc.c*), and the dBUG library and vector table.

Files *comp/libdbug.a* and *comp/vectors.o* are the dBUG library and vector table. The library and vector table are specific to the toolchain and CPU in use. See 4.8 dBUG Libraries for information about obtaining the appropriate library and vector table files.

4.4 CREATING THE BOARD SUPPORT PACKAGE

The first stage of creating the BSP is to do the minimum work necessary to allow dBUG to boot. Once dBUG is able to boot, features can be incrementally added.

4.4.1 Board Support Package Template Files

Accompanying dBUG is the source to a generic board support package which contains all the basic files and functions needed for a dBUG port. It is recommended that the generic board support package be copied into a new project directory as the basis for the new dBUG port. In most cases, the generic templates need only be completed with the board-specific details.

Alternatively, a completed dBUG BSP with similar features may serve well as a starting point.

4.4.2 Initial Board Support Package

The steps for completing the initial board support package are straightforward.

1. Set the LIBDBUGHOST environment variable. See 4.8 dBUG Libraries for more information.
2. Edit *config.h* and define the appropriate CPU. See *dss/src/include/cpu/cpu.h* for a complete list of supported processors. Do **NOT** define DBUG_NETWORK at this time.
3. Edit *board.h* to provide any necessary prototypes, data structures or definitions. Memory map and device definitions provided in this file often prove useful.
4. Edit *board.c* and provide the details to the required board-specific functions listed in Table 4-1.
5. Create any required CPU-specific functions. For example, the integrated processors supported by dBUG require board-specific initialization of the integrated peripherals. See Section 3 Writing dBUG Board Support Packages for details on the CPU-specific functions.
6. Edit the files *comp/make.cpu* and *comp/board.lnk* to accommodate the toolchain in use. Exact details for configuring the toolchain and linker files are beyond the scope of this document, and must be referred to the host toolchain documentation. Be careful to place ROM and RAM sections correctly!
7. Modify the top-level project makefile to invoke the correct subordinate makefile and define the correct output directory.
8. Modify the subordinate makefile and declare the macro BOARD appropriately; it must match a value used in *config.h*.

Upon completing the above steps, executing *make* in the project directory begins the compilation of the board support package.

4.5 DEBUGGING THE BOARD SUPPORT PACKAGE

After building the BSP, some debugging may be necessary. The two most problematic areas requiring debug are the initialization code and toolchain related issues.

4.5.1 dBUG Run-Time Entry Points

To aid the debugging of initialization code, it is useful to know the execution path of dBUG out of reset. The execution path at reset performs all basic initialization of the system.

The reset execution path as well as the other run-time execution paths are detailed below.

dBUG obtains control at three primary entry points:

- Reset
- General Exception (excluding interrupts)
- Interrupts

The Reset entry point is executed at board power-up, board hard reset, or the RESET command. The general code sequence executed is the following:

1. reset vector - The reset vector, located in *vectors.o*, points to asm_startmeup.
2. asm_startmeup - Located in *libdbug.a*, this code invalidates caches, disables interrupt, caching and address translation, and sets other CPU internal resources to a disabled, known state. Depending upon the processor, CPU-specific initialization code in the board support package is executed. When complete, main() is invoked.
3. main() - Located in *libdbug.a*, this routine performs the remaining initialization of the board and dBUG. This routine copies the vector table from ROM to RAM, copies initialized data (.data section) from ROM to RAM, and zeroes uninitialized data (.bss section). The following functions are then called, in sequence: board_init(), dbug_init(), board_init2(), uif_cmd_ver(), board_init3(), and finally mainloop().
4. board_init() - Located in *board.c*, this routine, at a minimum, initializes the dBUG console port.
5. dbug_init() - Located in *libdbug.a*, this routine performs the initialization of dBUG internal variables and resources.
6. board_init2() - Located in *board.c*, this routine performs activities that require dBUG resources (such as registering an interrupt handler), or activities prior to displaying the dBUG banner (such as displaying the amount of installed memory).
7. uif_cmd_ver() - Located in *libdbug.a*, this function displays the dBUG version banner.
8. board_init3() - Located in *board.c*, this routine performs any activities prior to entering the interactive dBUG> command prompt (such as booting an operating system or other system software).
9. mainloop() - Located in *libdbug.a*, this routine enters into an infinite loop which displays the dBUG> command prompt and processes user input.

The General Exception entry point is encountered during memory access errors, breakpoints, single instruction tracing, and other general exceptions. The general code sequence executed is the following:

1. exception vector - The vector, located in *vectors.o*, points to asm_exception_handler.
2. asm_exception_handler - Located in *libdbug.a*, this code flushes and disables caches, disables interrupts and address translation, and stores the register context. When complete, cpu_handler() is invoked with the exception number.
3. cpu_handler() - Located in *libdbug.a*, this routine handles the exception. In most cases, the exception number and context information is displayed, and control passed to mainloop(). However, some exceptions (software breakpoints, for example) may return from cpu_handler() to asm_exception_handler, at which point the context is restored and execution resumes.

The Interrupt entry point is executed upon detection of a CPU interrupt. These interrupts are generated primarily by peripheral devices and require servicing. The general code sequence executed is the following:

1. interrupt vector - The vector, located in *vectors.o*, points to asm_irq_handler.
2. asm_irq_handler - Located in *libdbug.a*, this code saves volatile registers (as per the calling convention/ABI) on the current stack, and invokes isr_execute_handler() with the interrupt number.
3. isr_execute_handler() - Located in *libdbug.a*, this routine searches the table of interrupt service routines, ISR, registered with dBUG. If a match is located, the ISR is invoked, and its return value (TRUE or FALSE) is returned to asm_irq_handler. If no match is found, FALSE is returned to asm_irq_handler.
4. If the return value from isr_execute_handler() is TRUE, indicating the interrupt was serviced, then the context is restored and execution resumes. If the return value is FALSE, the complete register context is saved, cpu_handler() is invoked to display the exception information, and control passed to mainloop().

If user code takes over any of these entry points, then it is quite possible that dBUG will not work properly, if at all.

4.5.2 Compiler/Toolchain Considerations

An important toolchain issue to understand is the run-time memory footprint. For most systems, dBUG executes from ROM or flash memory, and uses RAM starting at address 0x00000000. Table 4-2 illustrates the typical memory footprint for these systems.

Table 4-2. dBUG Run-Time Memory Footprint

SYMBOL	MEMORY SECTION	COMMENT
...	...	
__DATA_ROM	.data	dBUG's initialized data is stored in ROM, but copied to RAM at boot-time.
	.text	The executable code for dBUG.
...	...	
__USER_SPACE	User RAM	This portion of memory is deemed usable by downloaded user programs, normally located at 0x00010000.
__SP_INIT __SP_END	Stack	Stack space for dBUG.
__HEAP_END __HEAP_START	Heap	Heap space for dBUG.
__BSS_END __BSS_START	.bss	dBUG's un-initialized data. It is cleared to zero at boot-time.
__DATA_END __DATA_RAM	.data	dBUG's initialized data. It is copied from ROM to RAM at boot-time.
__VECTOR_RAM	Vector Table	CPU vector table for dBUG, normally located at 0x00000000.

The **.text** section contains the executable code for dBUG. The **.data** section contains initialized data which is stored in ROM, but copied to RAM at boot-time. The **.bss** section is the uninitialized data for dBUG that is zeroed at boot-time.

The symbol names in the left-hand column are defined in the linker script file, and evaluate to a 32-bit value representing the appropriate address. For example, __DATA_RAM is the address of the.data section in RAM.

In most systems, dBUG requires 128K of ROM and 64K of RAM. The ROM provides storage for dBUG's executable code and initialized data, while RAM contains a CPU vector table, the run-time initialized and uninitialized data, heap, and stack space. The actual amount of

ROM and RAM required will vary depending upon the features added to dBUG as well as the compiler in use.

To conserve RAM used by dBUG, declare all constant data with the C qualifier *const* or *static const*. Constant initialized data will remain in ROM, thus not requiring any RAM at run-time.

To take advantage of the CPU architecture, compilers may produce sections other than .text, .data, and .bss. The advantage of additional sections is that references to items located in these sections are normally very quick; requiring a single instruction with an embedded offset to access the item. The disadvantage is that at all times one or more CPU registers must contain a pointer to the additional sections; this poses a problem for dBUG. By the nature of the environment, dBUG cannot provide any protection between itself and downloaded code, thus the values of CPU registers can be changed at any time by the downloaded code. Therefore, to support additional sections, the CPU registers must be managed at every possible entry point into dBUG (entry points are not just exceptions, but include system calls, interrupt handlers and functions directly callable by downloaded code). While it is possible to provide the necessary management, this significantly increases the complexity of dBUG, while not necessarily guaranteeing its reliability. In short, limit the compiler-generated sections to .text, .data, and .bss sections; items located in these sections are referenced by the compiler with absolute addresses. Unless otherwise specified, all dBUG libraries are built with only these standard three sections. Consult the toolchain documentation for information about controlling the use of sections.

The dBUG libraries are built using compiler options to achieve specific goals (see the discussion immediately above). Therefore, the board support package **must** be built using the same compiler options. All makefiles for the libraries are contained in the *libdbug* project and contain the compiler options used for building the libraries.

When interfacing assembly routines with C routines, the appropriate application binary interface (ABI) must be used. The ABI defines the usage of CPU registers and how parameters are passed between functions. In general, dBUG uses the ABI as defined by System V Release 4 Unix, SVR4. Also, most toolchains use differing and incompatible assembly source file formats, which adds to the difficulty of using assembly source files.

The placement of the dBUG vector table is an important toolchain issue. The linker must place the dBUG vector table so that the CPU can access it at power-on reset. File *vectors.o* contains the dBUG vector table, and since the dBUG vector table is normally placed first by the linker, it is provided separately from *libdbug.a*.

4.6 ADDING FEATURES TO THE BOARD SUPPORT PACKAGE

Once the basic BSP is working, features and new commands are easily added to dBUG.

4.6.1 Adding Commands

dBUG provides a core set of commands for performing basic system debugging activities. The command set can be extended to suit the particular board or application needs.

When the user enters a command, two searches through the command table are performed in order to locate the command. The first search seeks an exact match on the user-specified command and a command name in the table. If this search fails, a second search is performed seeking a match on the shortened command names.

The board-specific file *cmds.c* contains the dBUG command table.

```
UIF_CMD UIF_CMDTAB[] =
{
    UIF_CMDS_ALL
    CPU_CMDS_ALL
};
const int UIF_NUM_CMD = UIF_CMDTAB_SIZE;
```

The core command set is inserted with the macro UIF_CMDS_ALL, defined in dbug.h, and any CPU-specific commands are inserted with the macro CPU_CMDS_ALL. Additional commands are placed in the table following these macros. File *dbug.h* defines the command table entry data structure.

```
typedef const struct
{
    char * cmd;             /* command name user types, i.e. GO */
    int    unique;          /* num chars to uniquely match      */
    int    min_args;        /* min num of args command accepts  */
    int    max_args;        /* max num of args command accepts  */
    int    flags;           /* command flags (repeat, hidden)   */
    void   (*func)(int, char **); /* actual function to call    */
    char * description;     /* brief description of command     */
    char * syntax;          /* syntax of command                */
} UIF_CMD;
```

Field cmd is the command name as it is typed on the command line. Command names are eight characters or less in length.

Field unique indicates the number of characters required to match for the short name of the command. This value must be greater than zero, and less than the length of the command name.

Field min_args indicates the minimum number of arguments the command requires. If the user specifies fewer arguments than this field indicates, an error message is produced and the command is not invoked. This field must be equal to or greater than zero.

Field max_args indicates the maximum number of arguments the command accepts. If the user specifies more arguments than this field indicates, an error message is produced and the command is not invoked. The value for this field must equal or exceed the value for min_args, and may not exceed UIF_MAX_ARGS.

Field flags is used to slightly modify the behavior of the command. Flag UIF_CMD_FLAG_REPEAT indicates that the command is capable of rapid repeat execution. This flag indicates that the user may enter the command once, and then press <Return> to invoke subsequent executions of this command, i.e. the TRACE command. Flag UIF_CMD_FLAG_HIDDEN prevents the command from being displayed in the HELP menu.

Field func is the function to invoke when a command line matches the command name and meets its argument requirements. Function func receives two arguments, the first is the number of tokens on the command line (there is always at least one: the command), and the second is a pointer to an array of pointers pointing to each token on the command line. This scheme is similar to the invocation of the C language main() function.

Field description is the verbal description of the command displayed in the HELP menu.

Finally, field syntax describes the command usage and options. This information is displayed in the HELP menu.

For examples, the core command entries are located in *dbug.h*.

4.6.2 Adding SET/SHOW Options

dBUG provides a core set of SET/SHOW options for configuring dBUG. The SET/SHOW option set can be extended to suit the particular needs of the board.

When the user enters the SHOW command, the setting for the particular option is displayed. If no option is specified, then all option values are displayed.

When the user enters the SET or SHOW command, two searches through the SET/SHOW option table are performed in order to locate the option. The first search seeks an exact match on the user-specified option and an option name in the table. If this search fails, a second search is performed seeking a match on the shortened option names.

The board-specific file *cmds.c* contains the dBUG SET/SHOW option table.

```
UIF_SETCMD UIF_SETCMDTAB[] =
{
    UIF_SETCMDS_ALL
    CPU_SETCMDS_ALL
};
const int UIF_NUM_SETCMD = UIF_SETCMDTAB_SIZE;
```

The core option set is inserted with the macro UIF_SETCMDS_ALL, defined in dbug.h, and any CPU-specific commands are inserted with the macro CPU_SETCMDS_ALL. Additional options are placed in the table following these macros. File *dbug.h* defines the option table entry data structure.

```
typedef const struct
{
    char *  option;
    int     unique;
    int     min_args;
    int     max_args;
    int     flags;
    void    (*func)(int, char **);
    char *  syntax;
} UIF_SETCMD;
```

Field option is the option name as it is typed on the SET/SHOW command line. Option names are eight characters or less in length.

Field unique indicates the number of characters required to match for the short name of the option. This value must be greater than zero, and less than the length of the option name.

Field min_args indicates the minimum number of arguments the option requires. The value for this field must be at least one. If the user specifies fewer arguments than this field indicates, an error message is produced. This field must be equal to or greater than zero.

Field max_args indicates the maximum number of arguments the option requires. If the user specifies more arguments than this field indicates, an error message is produced. The value for this field must equal or exceed the value for min_args, and may not exceed UIF_MAX_ARGS.

Field flags is used to slightly modify the behavior of the command. Flag UIF_CMD_FLAG_HIDDEN prevents the option from being displayed in the SHOW menu.

Field func is the function to invoke when a command line matches the option name and meets its argument requirements. Function func receives two arguments. The first is the number of tokens on the command line, and the second is a pointer to an array of pointers pointing to each token on the command line. This scheme is similar to the invocation of the C language main() function.

Finally, field syntax describes the option usage and values. This information is displayed by SET.

Both the SET and SHOW commands use func. The indication of which command (SET or SHOW) invoked func is indicated in its first argument. If the value of the first argument is zero, one or two, then SHOW command invoked func to display option settings. If the value is zero, then the SHOW command is displaying all option values. Otherwise, when the value is three or greater, SET invoked func.

For examples, the common option entries are located in *dbug.h*.

4.6.3 Adding TFTP Download Support

The steps necessary for utilizing the TFTP Ethernet download are more complex, due to the Ethernet driver that must be written.

1. Edit *config.h* and #define DBUG_NETWORK.
2. Edit *board.c* and provide the board-specific functions needed during a network download. These functions are listed in Table 4-3 and are detailed in Section 4.10 Optional Board-Specific Functions.
3. Write the Ethernet driver. Details on writing an Ethernet driver are beyond the scope of this document; consult documentation for the Ethernet card or chip set. However, Ethernet drivers in *dss/src/dbug/v2/dev* provide examples on using the dBUG resources available to the driver.
4. The download functions in *board.c* need modification to accommodate the Ethernet download path (variable *uif_dlio* indicates the download type is UIF_DLIO_NETWORK). Function board_dlio_init() must register the interrupt handler and initialize the Ethernet driver. Function board_dlio_getchar() needs to call tftp_in_char(). Function board_dlio_done() must de-install the interrupt handler().
5. Depending upon the interrupt scheme, the interrupt handler may need to explicitly clear the Ethernet interrupt. As such, the interrupt handler may be an intermediate function which clears the interrupt, and in turn invokes the real Ethernet driver interrupt handler.

Table 4-3. Optional Board-Specific Functions

FUNCTION	DESCRIPTION
board_dlio_filetype()	Determine download filetype
board_irq_enable()	Enable interrupts
board_irq_disable()	Disable interrupts
board_set_client()	Set board IP address
board_get_client()	Get board IP address
board_set_server()	Set TFTP server IP address
board_get_server()	Get TFTP server IP address
board_set_gateway()	Set gateway IP address
board_get_gateway()	Get gateway IP address
board_set_netmask()	Set IP netmask
board_get_netmask()	Get IP netmask
board_set_filename()	Set default download filename
board_get_filename()	Get default download filename
board_set_filetype()	Set default download file type
board_get_filetype()	Get default download file type

To allow these changes to take effect, perform a *make clean* followed by a *make*. (The definition of DBUG_NETWORK affects other conditional macros, thus requiring the *make clean* at least once.) Once built, the dBUG command DN performs the network download.

4.7 RESOURCES AVAILABLE TO THE BOARD SUPPORT PACKAGE

Many resources are available for use by board support packages. All of the following resources are defined in *dss/src/include/dbug.h*.

4.7.1 Standard C Library

dBUG uses and provides several functions in the standard C library. By providing these standard C library functions, one dependency on the host toolchain is eliminated. Consult ANSI C for information on these functions.

- isspace(), isalnum(), isdigit(), isupper()
- strcmp(), strncmp(), strcasecmp(), strncasecmp()
- strtoul(), strlen(), strcat(), strncat(), strcpy(), strncpy()
- memcpy(), memset()
- printf(), sprintf()

NOTE

Printf() and sprintf() currently do not support floating point formats, and %b indicates a binary format.

4.7.2 User Interface Resources

Certain routines in the User Interface are available for BSPs that implement new commands or obtain user input. Common messages are available as well.

- COPYRIGHT - dBUG copyright banner message.
- HELPMSG - Help banner.
- INVARG - Useful for error messages, contains “Error: Invalid argument: %s\n”.
- INVCMD - Contains “Error: Invalid command: %s\n”.
- INVREG - Contains “Error: Invalid register: %s\n”.
- INVALUE - Contains “Error: Invalid value: %s\n”.
- UIF_MAX_ARGS - This value is the maximum number of arguments allowed on the command line.
- UIF_MAX_LINE - This value is the maximum length of command line input.
- UIF_VER_MAJOR - This value indicates the major revision of the common user interface features.
- UIF_VER_MINOR - This value indicates the minor revision of the user interface.
- BASE - This variable indicates the user’s preference for converting strings to numbers.
- pause() - Function for providing rudimentary display paging.
- get_line() - Function for obtaining command line input.
- make_argv() - Function for parsing command line input into tokens.

- get_value() - Function to convert a string or symbol name into a 32-bit value.

Additional details for the functions is provided in 4.11 dBUG Internal Functions.

4.7.3 CPU-Specific Resources

Certain routines and information in the CPU-specific portion are available for BSPs.

- CPU_STR - CPU name.
- CPU_VER_MAJOR - CPU-specific code major revision.
- CPU_VER_MINOR - CPU-specific code minor revision.
- context - This global data structure holds a copy of the CPU register context.

Many other functions exist which are used from within the User Interface, but are not available to board support packages.

See Section 3 CPU-Specific Information for details on additional services provided by the CPU-specific component.

4.7.4 Download Resources

The implementation for downloading files utilizes a simple byte-stream approach. During the download, board_dlio_getchar() is called to return the next byte in the data stream. However, the data stream can originate from a variety of sources. The source of the download data is set in the appropriate user command and the download process initiated. dBUG directly supports download via the console port, typically a serial port, or from an Ethernet network using TFTP.

- *uif_dlio* - this variable indicates the source of the download data stream, typically either UIF_DLIO_CONSOLE or UIF_DLIO_NETWORK.
- UIF_DLIO_CONSOLE - The data stream is obtained from the console port.
- UIF_DLIO_NETWORK - The data stream obtained via TFTP from the network.

When the download source is the network, dBUG processes the download data stream to accommodate ELF, COFF, S-Record and binary files. The filetype (ELF, COFF, S-Record or binary) is indicated on the DN command line, or can be derived from the download filename extension.

- *.elf - Download filetype is UIF_DLIO_ELF.
- *.coff - Download filetype is UIF_DLIO_COFF.
- *.srec - Download filetype is UIF_DLIO_SREC.
- *.bin - Download filetype is UIF_DLIO_IMAGE.

Additional filename extensions can be associated with one of the above filetypes. The board-specific function board_dlio_filetype() returns one of the above filetypes, or UIF_DLIO_UNKNOWN. This function is documented in 4.10 Optional Board-Specific Functions.

As an example, these are the general steps for downloading from a parallel port.

1. Define the new download stream source, i.e. UIF_DLIO_PARALLEL in *board.h* (do not conflict with the sources defined in *dbug.h*).
2. Modify the functions board_dlio_init(), board_dlio_getchar() and board_dlio_done() to accommodate the new stream source.
3. Create the parallel port driver that is invoked from within board_dlio_init(), board_dlio_getchar() and board_dlio_done().
4. Create a new user command, i.e. DP, that sets *uif_dlio* to the value UIF_DLIO_PARALLEL, calls board_dlio_init(), then calls download_srecord() to perform an S-record download, and finishes by calling board_dlio_done().
5. Add the new command to the dBUG command set.

Once dBUG is rebuilt, the new DP command can be used to download S-records from the system's parallel port.

4.7.5 Interrupt Handling Resources

dBUG provides a method for hooking CPU interrupts. By registering an interrupt handler with dBUG, CPU register context save and restore operations are performed by dBUG, thus relieving the user of the need to manage context preservation.

- isr_register_handler() - This function installs an interrupt service routine.
- isr_remove_handler() - This function removes a previously installed interrupt service routine.
- ISR_DBUG_ISR - An argument to isr_register_handler(), this flag gives the handler priority over other handlers installed on the same interrupt vector with ISR_USER_ISR.
- ISR_USER_ISR - An argument to isr_register_handler(), this flag indicates a lower priority handler for the interrupt vector.

dBUG maintains a simple list of registered handlers. When an interrupt occurs, dBUG first examines the list for a match on the interrupt number and ISR_DBUG_ISR. If a match is found, the handler is invoked. If the handler returns FALSE, indicating that the interrupt was not serviced, the search continues for another match on the interrupt handler and ISR_DBUG_ISR. If the handler returns TRUE, then the search on ISR_DBUG_ISR stops. When dBUG completes its search for the interrupt number and ISR_DBUG_ISR, it then performs a search for the interrupt number and ISR_USER_ISR in the same fashion.

While dBUG itself is executing, all interrupts are disabled. For the CPU to recognize an interrupt and invoke any interrupt service routine, interrupts must explicitly be enabled. Furthermore, dBUG disables interrupts at all exception entry points, except handled interrupts. Note that default settings for CPU control registers enable interrupts when executing user code via the GO and GT commands.

Additional information on these functions is provided in 4.11 dBUG Internal Functions.

4.7.6 Miscellaneous Resources

The following are used by dBUG, and are available for general use.

- TRUE - Evaluates to 1.
- FALSE - Evaluates to 0.
- NULL - Evaluates to 0.
- __USER_SPACE - Address of memory available for general use (and not used by dBUG).

4.8 DBUG LIBRARIES

dBUG is provided in object format as a library of routines which can be linked with the board support package to create a usable debugger. A dBUG library contains the common User Interface component as well as the CPU-specific component.

4.8.1 LIBDBUGHOST Environment Variable

The LIBDBUGHOST environment variable is used to indicate on which host dBUG is being built. The value for LIBDBUGHOST must be one of the supported hosts.

- SUNS - Sun Solaris Unix workstation
- WIN32 - Windows 95/NT personal computer

Binary files created on one host type are normally not compatible with binaries created on a different host type. For example, a dBUG library built on a Sun Solaris machine is not usable for building dBUG on a Windows machine. As a result, separate dBUG libraries must be provided for each host. Thus the environment variable LIBDBUGHOST indicates the host and is used to correctly locate binary (and other host-specific) files for building dBUG.

When creating the dBUG libraries, the LIBDBUGHOST environment variable is used to locate the appropriate compiler as well as the destination directory for the resulting dBUG library.

When creating board support packages, the LIBDBUGHOST environment variable is used to select the proper binary files for *libdbug.a* and *vectors.o*. The environment variable is also used in the makefiles to locate the appropriate compilers and their components.

In addition, the environment variable is used to determine host-specific information. For example, the files *libdbug/src/dbug/comp/compilers.** contain the paths to the supported toolchains on the appropriate host.

It is recommended that the value for the LIBDBUGHOST environment variable be set in an appropriate startup file.

All libraries are located in the *libdbug* project. The directory arrangement reflects the supported processors and toolchains, as illustrated in Figure 4-3.

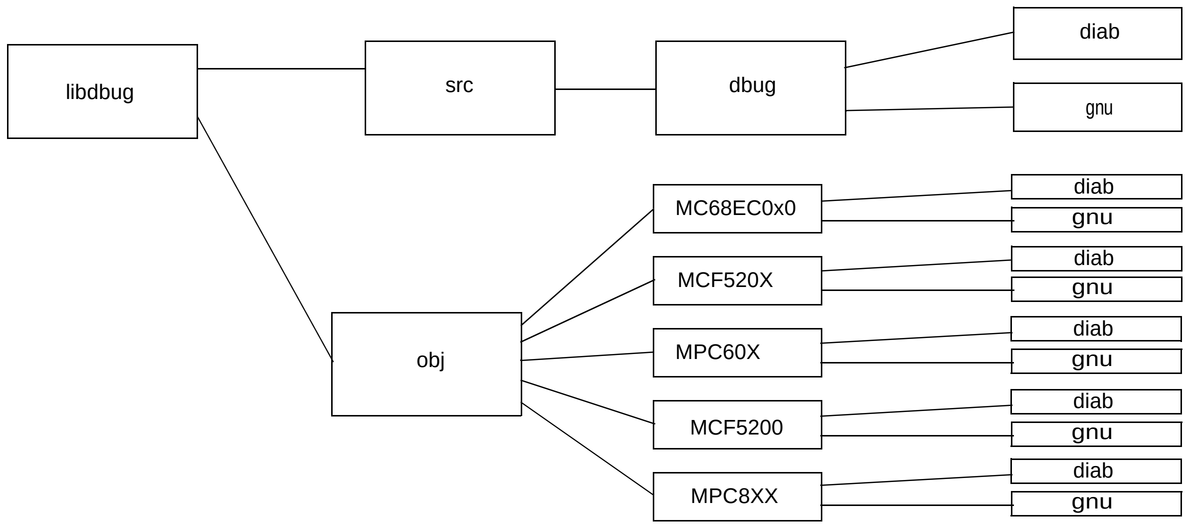


Figure 4-3. dBUG Libraries

The libraries are arranged by target processor and then the toolchain vendor and version. For example, the dBUG library located in *libdbug/obj/mcf5200/diab40bE/SUNS* was created with a Sun Solaris-hosted Diab Data version 4.0b toolchain targeted for the ColdFire MCF5200 processor. The trailing ‘E’ in the toolchain name indicates an ELF format library, whereas ‘C’ indicates COFF format.

More information about these toolchains as it relates to the dBUG effort is provided below.

4.8.2 Diab Data, Inc.

Diab's suite of tools includes cross-compilers for MC68000, ColdFire and PowerPC. dBUG has been built using the following versions of Diab's compilers:

- 3.6
- 3.7
- 4.0
- 4.1

The files in *libdbug/src/dbug/diab* illustrate which compiler options are used for building dBUG.

See http://www.ddi.com for more information on Diab's products.

4.8.3 GNU C

The popular GNU C compiler can also be configured as a cross-compiler for MC68000, ColdFire and PowerPC targets. dBUG has been built using the following versions of GNU C.

- Gcc 2.7.2.2 + patch
- Gcc 2.7.2.3 + patch
- Gcc 2.8.0
- Gcc 2.8.1

The GNU assembler and linker are needed as well. The following versions have been used to build dBUG.

- Binutils 2.8.1
- Binutils 2.9
- Binutils 2.9.1

dBUG libraries are available for these versions of the compiler, but none are tested. Every now and then, bugs are discovered in the toolchain.

For MC68000 and ColdFire, the GNU cross-compiler is configured as a **m68k-coff** target for use with the GNU assembler and linker. This configuration produces COFF object and executable files.

For PowerPC, the GNU cross-compiler is configured as a **powerpc-eabi** target for use with the GNU assembler and linker. This configuration produces ELF object and executable files.

The files in *libdbug/src/dbug/gnu* illustrate which compiler options are used for building dBUG.

See http://www.gnu.org for more information on GNU C and other products. GNU C and Binutils distributions are located at ftp://prep.ai.mit.edu, and patches are available at ftp://ftp.cygnus.com/pub/embedded.

4.8.4 Experimental GNU Compiler System

The Experimental GNU Compiler System, EGCS, is a derivative of the GNU C compiler. It was created by a group of developers who were unhappy with the infrequent releases of GNU C. These developers now actively work on the EGCS distribution and produce snapshots weekly and more frequent releases of the compiler.

For all practical purposes, the EGCS compiler is a newer version of the GNU C compiler. The EGCS compiler can also be configured as a cross-compiler for MC68000, ColdFire and PowerPC targets. dBUG has been built using the following versions of EGCS.

- Egcs 1.0
- Egcs 1.0.1
- Egcs 1.0.2
- Egcs 1.0.3
- Egcs 1.1.1
- Egcs 1.1.2

The GNU assembler and linker are needed as well. The following versions have been used to build dBUG.

- Binutils 2.8.1
- Binutils 2.9
- Binutils 2.9.1

dBUG libraries are available for these versions of the compiler, but none are tested. Be aware that the EGCS effort is a work in progress. As such, bugs are found, features broken and bugs introduced; all in the constant effort to improve the compiler.

For MC68000 and ColdFire, the EGCS cross-compiler is configured as a **m68k-coff** target for use with the GNU assembler and linker. This configuration produces COFF object and executable files.

For PowerPC, the EGCS cross-compiler is configured as a **powerpc-eabi** target for use with the GNU assembler and linker. This configuration produces ELF object and executable files.

The files in *libdbug/src/dbug/gnu* illustrate which compiler options are used for building dBUG.

See http://egcs.cygnus.com for more information on EGCS. Distributions are located at ftp://egcs.cygnus.com/pub/egcs and Binutils distributions are located at ftp://prep.ai.mit.edu.

4.9 BOARD-SPECIFIC FUNCTIONS

The following functions are needed for board support packages. These functions typically reside in the file *board.c*.

board_init() Board Initialization Function

Syntax:

void board_init (void);

Description:

This is the first of three board initialization functions invoked by dBUG. This function performs the majority of board initialization.

Activities that must be completed by board_init() include: 1) dBUG console port initialization, and 2) other peripheral initialization.

Until the dBUG console port is initialized, printf() will not work.

When board_init() returns, dBUG performs internal initialization.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_init2()
board_init3()

board_init2() Board Initialization Function

Syntax:

```
void board_init2 (void);
```

Description:

This function is the second of three board initialization functions. Any initialization of the board that draws on internal resources of dBUG may be performed here.

If not performed in board_init(), the console port used by dBUG must be initialized in this function. Upon returning from board_init2(), dBUG invokes printf() in displaying the start-up banner. Until the dBUG console port is initialized, printf() will not work.

At this point, initialization of dBUG is complete. If necessary, hooks can be placed in this function to perform operating system bootstrap or other system features.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_init()
board_init3()

board_init3() Board Initialization Function

Syntax:

```
void board_init3 (void);
```

Description:

This function is the third of three board initialization functions. Any initialization of the board that draws on internal resources of dBUG may be performed here.

Upon returning from board_init3(), dBUG displays the help message and the dBUG command prompt, dBUG>.

If necessary, hooks can be placed in this function to perform operating system bootstrap or other system features.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_init()
board_init2()

board_getchar() Character input

Syntax:

```
int board_getchar (void);
```

Description:

This function obtains the character available on the dBUG console port.

This function must poll until a character is available.

Parameters:

None.

Return values:

Character input from dBUG console port.

Errors:

None.

See Also:

board_putchar()
board_getchar_present()

board_putchar() Character output

Syntax:

void board_putchar (int ch);

Description:

This function outputs a character on the dBUG console port.

This function must not return until the character is output.

This function is called directly by printf().

Parameters:

ch	The character to output.

Return values:

None.

Errors:

None.

See Also:

board_getchar()
board_putchar_flush()

board_getchar_present() Test for character input

Syntax:

```
int board_getchar_present (void);
```

Description:

This function tests whether a character is available on the dBUG console port.

This function does NOT poll until a character is available, it merely tests for the presence of a character.

Parameters:

None.

Return values:

TRUE	Character is available.
FALSE	Character is not available.

Errors:

None.

See Also:

board_getchar()

board_putchar_flush() Flush character output

Syntax:

void board_putchar_flush (void);

Description:

This function is called prior to displaying the dBUG> prompt in order to flush output characters on the dBUG console port.

For dBUG console ports which are serial ports, this function is typically empty.

Parameters:

None.

Return values:

TRUE	Character is available.
FALSE	Character is not available.

Errors:

None.

See Also:

board_putchar()

board_dlio_getchar() Download character input

Syntax:

```
int board_dlio_getchar (void);
```

Description:

This function is called during a download to obtain the next data byte.

The global variable *uif_dlio* indicates the type of download being performed. When the DL command is invoked, *uif_dlio* indicates UIF_DLIO_CONSOLE. When the DN command is invoked, *uif_dlio* indicates UIF_DLIO_NETWORK.

For console downloads, this function returns the value obtained from board_getchar(). For network downloads, this function returns the value obtained from tftp_in_char().

Parameters:

None.

Return values:

Next character.

Errors:

None.

See Also:

board_dlio_done()
board_dlio_init()
board_dlio_vda()

board_dlio_init() Download initialization

Syntax:

int board_dlio_init (void);

Description:

This function is called prior to performing a download to perform initialization or activities needed to assist the download.

The global variable *uif_dlio* indicates the type of download being performed. When the DL command is invoked, *uif_dlio* indicates UIF_DLIO_CONSOLE. When the DN command is invoked, *uif_dlio* indicates UIF_DLIO_NETWORK.

For console downloads, typically there are no tasks for this function to perform. However, for network downloads, this function is required to perform two tasks: 1) register an interrupt service routine, and 2) initialize the network device.

For both console and network downloads, sometimes it is useful to enable instruction (but not data) caching at this time.

Parameters:

None.

Return values:

TRUE	Download can proceed.
FALSE	Download can not proceed.

Errors:

None.

See Also:

board_dlio_done()
board_dlio_getchar()
board_dlio_vda()

board_dlio_vda() Download valid address

Syntax:

```
int board_dlio_vda (ADDRESS addr);
```

Description:

This function is called during a download to determine if an address is a valid download address.

A given 32-bit address is not always a valid address for placing download data. An address that points to the dBUG reserved space, ROM, or I/O or un-populated RAM is an invalid address.

At a minimum, this function compares the provided address against the known dBUG reserved space and system RAM to determine if addr can be used to store download data.

Parameters:

addr	Address at which to download.

Return values:

TRUE	addr is valid address at which to download.
FALSE	addr is not a valid address at which to download.

Errors:

None.

See Also:

board_dlio_done()
board_dlio_init()
board_dlio_getchar()

board_dlio_done() Download completion

Syntax:

```
void board_dlio_done (void);
```

Description:

This function is called after completing a download to stop the download process.

The global variable *uif_dlio* indicates the type of download performed. When the DL command is invoked, *uif_dlio* indicates UIF_DLIO_CONSOLE. When the DN command is invoked, *uif_dlio* indicates UIF_DLIO_NETWORK.

For console downloads, typically there are no tasks for this function to perform. However, for network downloads, this function is required to de-register the interrupt service routine and graceful turn off the Ethernet device.

If instruction caching is enabled during the download, then this function should disable caching.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_dlio_init()
board_dlio_getchar()
board_dlio_vda()

board_get_baud() Get baud rate of dBUG port

Syntax:

```
int board_get_baud (void);
```

Description:

This function is called to obtain the baud rate of the dBUG console port. The value returned by this function is used in configuring the console port at boot time.

This function is also invoked by the SHOW BAUD command.

To fix the baud rate at a particular value, simply return the desired value and make board_set_baud() do nothing.

If possible, the baud rate value should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

None.

Return values:

Typical values are 9600, 19200 and 38400.

Errors:

None.

See Also:

board_set_baud()

board_set_baud() Set baud rate of dBUG port

Syntax:

void board_set_baud (int baud);

Description:

This function is called to set the baud rate of the dBUG console port. Because this value is used in configuring the console port at boot time, it is helpful if this value is stored in persistent memory, i.e. non-volatile RAM.

This function is invoked by the SET BAUD command.

If a fixed baud rate is being used, then this function should be empty.

If possible, the baud rate value should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

baud	Typical values are 9600, 19200 and 38400.

Return values:

None.

Errors:

None.

See Also:

board_get_baud()

board_reset() Board reset

Syntax:

```
void board_reset (void);
```

Description:

This function is invoked by the RESET command to reset the board.

This function contains code for a software-initiated reset. If no such mechanism exists, then this function is empty and dBUG executes the same code sequence as if a hard reset occurred.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

None.

4.10 OPTIONAL BOARD-SPECIFIC FUNCTIONS

These functions are needed only if the TFTP download feature is utilized.

board_dlio_filetype() Determine download filetype

Syntax:

```
int board_dlio_filetype (char *fn, char *ext);
```

Description:

This function determines the download filetype (ELF, COFF, S-Record or Binary) from the filename. dBUG examines the extension (that part of the filename following the period, if one exists) to determine the download filetype if none is specified on the DN command line.

Parameters:

fn	Pointer to the string containing the download filename.
ext	Pointer to the filename extension

Return values:

UIF_DLIO_UNKNOWN	Download filetype is not know.
UIF_DLIO_ELF	Download filetype is ELF.
UIF_DLIO_COFF	Download filetype is COFF.
UIF_DLIO_SREC	Download filetype is S-Record.
UIF_DLIO_IMAGE	Download filetype is binary data.

Errors:

None.

See Also:

None.

board_irq_enable() Enable board interrupts

Syntax:

```
void board_irq_enable (void);
```

Description:

This function is used to enable board interrupts during the network download. It is used in conjunction with board_irq_disable() to delineate critical section processing during the download.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_irq_disable()

board_irq_disable() Disable board interrupts

Syntax:

void board_irq_disable (void);

Description:

This function is used to disable board interrupts during the network download. It is used in conjunction with board_irq_enable() to delineate critical section processing during the download.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_irq_enable()

board_set_client() Set board IP address

Syntax:

```
void board_set_client (uint8 *ipaddr);
```

Description:

This function is used to store the Internet Protocol (IP) address of the board in persistent storage. This function is invoked when the user enters the SET CLIENT command.

If possible, the client IP should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to the 4-byte IP address.

Return values:

None.

Errors:

None.

See Also:

board_get_client()

board_get_client() Get board IP address

Syntax:

```
uint8 * board_get_client (uint8 *ipaddr);
```

Description:

This function is used to retrieve the IP address of the board from persistent storage. This function is invoked when the user enters the SHOW CLIENT command, and by the DN command.

If possible, the client IP should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to buffer for copying the 4-byte IP address.

Return values:

Pointer to the 4-byte IP address.

Errors:

None.

See Also:

board_set_client()

board_set_server() Set server IP address

Syntax:

```
void board_set_server (uint8 *ipaddr);
```

Description:

This function is used to store the IP address of the server in persistent storage. This function is invoked when the user enters the SET SERVER command.

If possible, the server IP should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to the 4-byte IP address.

Return values:

None.

Errors:

None.

See Also:

board_get_server()

board_get_server() Get server IP address

Syntax:

```
uint8 * board_get_server (uint8 *ipaddr);
```

Description:

This function is used to retrieve the IP address of the server from persistent storage. This function is invoked when the user enters the SHOW SERVER command, and by the DN command.

If possible, the server IP should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to buffer for copying the 4-byte IP address.

Return values:

Pointer to the 4-byte IP address.

Errors:

None.

See Also:

board_set_server()

board_set_gateway() Set gateway IP address

Syntax:

```
void board_set_gateway (uint8 *ipaddr);
```

Description:

This function is used to store the IP address of the gateway in persistent storage. This function is invoked when the user enters the SET GATEWAY command.

If possible, the gateway IP should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to the 4-byte IP address.

Return values:

None.

Errors:

None.

See Also:

board_get_gateway()

board_get_gateway() — Get gateway IP address

Syntax:

```
uint8 * board_get_gateway (uint8 *ipaddr);
```

Description:

This function is used to retrieve the IP address of the gateway from persistent storage. This function is invoked when the user enters the SHOW GATEWAY command, and by the DN command.

If possible, the gateway IP should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to buffer for copying the 4-byte IP address.

Return values:

Pointer to the 4-byte IP address.

Errors:

None.

See Also:

board_set_gateway()

board_set_netmask() Set IP netmask

Syntax:

```
void board_set_netmask (uint8 *ipmask);
```

Description:

This function is used to store the IP netmask in persistent storage. This function is invoked when the user enters the SET NETMASK command.

If possible, the IP netmask should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

ipmask	Pointer to the 4-byte IP netmask.

Return values:

None.

Errors:

None.

See Also:

board_get_netmask()

board_get_netmask() Get IP netmask

Syntax:

```
uint8 * board_get_netmask (uint8 *ipmask);
```

Description:

This function is used to retrieve the IP netmask from persistent storage. This function is invoked when the user enters the SHOW NETMASK command, and by the DN command.

If possible, the IP netmask should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

ipmask	Pointer to buffer for copying the 4-byte IP netmask.

Return values:

Pointer to the 4-byte IP netmask.

Errors:

None.

See Also:

board_set_netmask()

board_set_filename() Set default download filename

Syntax:

```
void board_set_filename (char *filename);
```

Description:

This function is used to store the default filename for network downloads in persistent storage. This function is invoked when the user enters the SET FILENAME command.

If possible, the filename should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

filename	Pointer to the filename.

Return values:

None.

Errors:

None.

See Also:

board_get_filename()

board_get_filename() Get default download filename

Syntax:

```
char * board_get_filename (char *filename);
```

Description:

This function is used to retrieve the default network download filename from persistent storage. This function is invoked when the user enters the SHOW FILENAME command, and by the DN command.

If possible, the filename should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

filename	Pointer to buffer for copying the filename.

Return values:

Pointer to the filename.

Errors:

None.

See Also:

board_set_filename()

board_set_filetype() Set default download filetype

Syntax:

void board_set_filetype (int filetype);

Description:

This function is used to set the default filetype for network downloads. This function is invoked when the user enters the SET FILETYPE command.

If possible, the filetype should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

filetype	The download filetype, either UIF_DLIO_SREC, UIF_DLIO_ELF, UIF_DLIO_COFF or UIF_DLIO_IMAGE.

Return values:

None.

Errors:

None.

See Also:

board_get_filetype()

board_get_filetype() Get default download filetype

Syntax:

```
int board_get_filetype (void);
```

Description:

This function is used to retrieve the default network download filetype. This function is invoked when the user enters the SHOW FILETYPE command, and by the DN command.

If possible, the filetype should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

None.

Return values:

The download filetype, either UIF_DLIO_ELF, UIF_DLIO_COFF, UIF_DLIO_SREC or UIF_DLIO_IMAGE.

Errors:

None.

See Also:

board_set_filetype()

4.11 DBUG INTERNAL FUNCTIONS

These functions are provided in libdbug.a and are usable by the board support package.

pause() Simple console pagination

Syntax:

int pause (int *rows);

Description:

This function provides a simple method for console pagination. On each invocation of this function, the value rows is incremented by one. When the value of rows surpasses 21, the banner is displayed and awaits user input to continue.

```
Press <ENTER> to continue.
```

This function is **NOT** invoked automatically by printf(); instead, this function must be explicitly invoked when pagination is desired. To use this function, the user must initialize a local integer variable to the value zero, then after displaying a single line of output, invoke pause(). The variable rows should track the number of lines of output displayed to the console. When the user selects <Enter> to continue, the variable rows is reset to zero.

The user may select <Enter> to continue, and q or Q or <Ctrl> C to indicate a desire to abort the display.

Parameters:

rows	Integer value indicates current number of lines of output displaying on the console.

Return values:

TRUE	User selected q or Q or <Ctrl> C.
FALSE	User selected <Enter>.

Errors:

None.

See Also:

None.

get_line() Console input

Syntax:

```
char * get_line (char *line);
```

Description:

This function obtains a line of input from the dBUG console and places it into the user-supplied character buffer line. The backspace and delete keys provide a rub-out feature, the only editing capability. This function returns when the user has pressed the <Enter> key. The buffer is properly terminated.

The character buffer must be of size UIF_MAX_LINE or greater.

Parameters:

line	Pointer to the character buffer for the user input.

Return values:

This function returns a pointer to the head of the character buffer. This value is equivalent to the value of line.

Errors:

None.

See Also:

make_argv()

make_argv() Parse string into tokens

Syntax:

```
int make_argv (char *line, char *argv[]);
```

Description:

This function parses the NULL-terminated string line into tokens, and places pointers to the resulting tokens into argv[]. This function is commonly used to process user input.

Tokens are delineated by white space. As the function scans the string, it places the NULL character \0 into the string in place of white space. In doing so, the token becomes a properly NULL-terminated string. The number of tokens parsed is returned.

The token list argv[] must be a minimum size UIF_MAX_ARGS plus one for a NULL terminated list. The token list is NULL terminated upon completing the scan.

This function modifies the original string.

Parameters:

line	Pointer to the character buffer to be processed. This buffer will be modified.
argv	Pointer to the array of character pointers which point to the tokens. This array is NULL terminated.

Return values:

This function returns the number of tokens parsed.

Errors:

None.

See Also:

get_line()

get_value() Convert string to number

Syntax:

```
uint32 get_value (char *str, int *success, int base);
```

Description:

This function converts the string str into a 32-bit unsigned number.

The function first attempts to locate the string in the symbol table. If a match is found, the value of the symbol is returned.

Otherwise the function converts the string according to radix base. The radix is a value 2 for binary, 8 for octal, 10 for decimal or 16 for hexadecimal. The value 0 for radix indicates that get_value() should determine the radix by examining the string for radix indicators.

Parameters:

str	Pointer to the character string to be converted.
success	Pointer to an integer. This variable indicates whether or not the conversion encountered errors.
base	The radix for converting the string. This value must be between 0 and 16 (inclusive).

Return values:

If no errors are encountered, this function returns a 32-bit unsigned value and success is TRUE. If errors are detected, the value zero is returned, and success is FALSE.

Errors:

For a given radix, certain characters are valid. Hexadecimal notation, for example, allows the letters ‘0’ through ‘9’ and ‘A’ through ‘F’, but not the letter ‘G’. If an illegal character is encountered, then success contains FALSE.

See Also:

strtoul()

isr_register_handler() Install an Interrupt Service Routine

Syntax:

int isr_register_handler (int type, int vector,int (*handler)(void *arg1, void *arg2), void *arg1, void *arg2);

Description:

This function installs an interrupt service routine (ISR) for the indicated vector. The type provides a relative priority for handlers which may be installed on the same vector: type ISR_DBUG_ISR is serviced prior to ISR_USER_ISR. This scheme allows dBUG to prioritize internal interrupt handlers over user-installed interrupt handlers.

When an interrupt occurs, dBUG saves the registers on the stack and searches the list of registered interrupt service routines. If an ISR for the appropriate vector is located, dBUG executes the ISR by invoking

```
handler(arg1, arg2);
```

dBUG examines the return value of the ISR to determine whether the interrupt was successfully serviced. If the return value is TRUE, then dBUG restores the registers and continues execution. Otherwise, dBUG dumps the register set to the console and displays the dBUG> prompt.

NOTE: While any vector can be passed to this routine, the ISR will only be invoked for vectors that are actually CPU interrupt vectors. For example, installing an interrupt handler for the same vector as the bus exception vector will never be invoked because the exception handling for the bus exception never examines the list of interrupt service routines.

Parameters:

type	Relative priority type of interrupt: ISR_DBUG_ISR or ISR_USER_ISR.
vector	CPU-specific vector number for the interrupt.
handler	The address of the interrupt service routine.
arg1	Pointer to an implementation-specific value or data structure.
arg2	Pointer to an implementation-specific value or data structure.

Return values:

If TRUE is returned, the handler was successfully installed. Otherwise, the handler was not installed.

Errors:

None.

See Also:

isr_remove_handler()

isr_remove_handler() Remove an Interrupt Service Routine

Syntax:

```
int isr_remove_handler (int (*handler)(void *arg1, void *arg2));
```

Description:

This function removes an interrupt handler for handler previously installed with isr_register_handler().

Parameters:

handler	Interrupt service routine address.

Return values:

If TRUE is returned, the handler was successfully un-installed.

Errors:

None.

See Also:

isr_register_handler()