MPC8BUG USER'S MANUAL

APRIL 27, 1996

REVISION 0.6

CHAPTER 1
MPC8BUG OVERVIEW

1.1 INTRODUCTION

This document describes the monitor debugger for the Motorola MPC860 PowerQUICC processor device. This debugger runs on a host machince such as an IBM-PC or Sun and controls the PowerQUICC on the 860 Application Development System (860ADS) board.

The generic name for a family of host-based PowerPC debuggers is "MPCbug". The name for this debugger as specifically implemented for the MPC8xx series of devices (such as the MPC860) is "MPC8bug". In this document both terms are used since most of the features of MPC8bug apply to the more generic "MPCbug".

1.2 FEATURES

- Works through the debug port on the MPC860.
- Communication between the host and the ADS board is through an ADI card.
- Hosted on Sun Sparcstations and IBM-PC (486 and higher) compatible platforms.
- User interface consists of mnemonic type commands.
- Allows the user to modify and display board memory and MPC860 registers in a variety of formats.
- An on-line assembler is provided (extended mnemonics not included)
- Provides an interface to the MPC860 breakpoint logic.
- Loads programs in COFF, XCOFF, ELF, or Srecords format.
- Provides aliases, command files, automatic initialization command file and log files.
- Disassembled instructions provide symbolic information (which can also be used in address expressions).
- Supports system calls, program I/O and command line arguments.

CHAPTER 2
STARTING THE DEBUGGER

2.1 COMMAND LINE ARGUMENTS

mpc8bug ADI ADS [-nosem] [object_file] [-c command_file] [-o log_file]

ADI- a number in the range 1-3.
On a Sparcstation: The number of the SBus slot of the ADI card.
On a PC: address of the port of the ADI card divided by 0x100.

ADS- address of ADS board (0-7). (determined by dip switches on the board.)

nosem - suppress semaphore allocation upon activation. (By default, allows only one debugger to be activated per each ADI port.) This argument is not applicable in IBM-compatible PCs.

object_file - name of a program to load.

command_file - name of a file containing debugger commands.

log_file - name of a file to which session history will be logged.

Example: mpc8bug 1 7

IMPORTANT NOTE: If this is the first use of the debugger since the 860ADS has been powered up from a cold start, the first command issued to the debugger should be a hard reset: "RESET :H".

NOTE: It is suggested not to use the ABORT or RESET buttons on the 860ADS board, but rather use the abort and reset commands provided within this debugger. If any buttons on the board are pressed, the first command issued to the debugger should be a hard reset: "RESET :H". This avoids any synchronization problems between the debugger and the 860ADS board.

2.2 PC CONFIGURATION

The PC version requires:

- A 486/586 processor, at least 4MB of memory, Dos6.0 or later.
- The frequency of the ISA bus in the BIOS setup must be set to no more than 8Mhz to ensure correct operation of the ADI card. Otherwise mpc8bug will not work.
- The length of the flat cable should not exceed 1 meter.

CHAPTER 3
MPC860 INITIALIZATION

3.1 INITIALIZATION VALUES

When mpc8bug is activated, it resets the ADS board. After reset, assuming the board is populated with the default memories, the following register initializations take place:

Core registers:

MSR and SRR1 = 0x00001002
DER = 0xffe74c0f
IMMR = 0x0220XXXX
ICTRL = 0x00000007

SIU registers:

SIUMCR = The old value bitwise OR'ed with 0x00032640
SYPCR = 0xffffff88
TBSCR = 0x0002
RTCSC = 0x0102
PISCR = 0x0002

The UPM and Memory Controller are initialized only after a hard reset.
Memory Controller:
OR0 = 0xffe00954
OR1 = 0xffff8110
OR2 = 0xffc00800
BR0 = 0x02800001
BR1 = 0x02100001
BR2 = 0x00000081
MPTPR = 0x0800
MAMR = 0x9ca21114

UPM contents for the default ADS memory configuration:

Read Single Beat Cycle:
```
00:  8fffec24 0fffec04 0cffec04 00ffec04
04:  00ffec00 37ffec47 ffffffff ffffffff
```

Read Burst Cycle:
```
08:  0fffec24 0fffec04 08ffec04 00ffec0c
```

```
0c:  03ffec00 00ffec44 00ffcc08 0cffcc44
10:  00ffec0c 03ffec00 00ffec44 00ffcc00
14:  3fffc847 3fffec47 ffffffff ffffffff

Write Single Beat Cycle:
18:  8fafcc24 0fafcc04 0cafcc00 11bfcc47
1c:  c0ffcc84 ffffffff ffffffff ffffffff

Write Burst Cycle:
20:  8fafcc24 0fafcc04 0cafcc00 03afcc4c
24:  0cafcc00 03afcc4c 0cafcc00 03afcc4c
28:  0cafcc00 33bfcc4f ffffffff ffffffff
2c:  ffffffff ffffffff ffffffff ffffffff

Periodic Timer Expired:
30:  c0ffcc84 00ffcc04 07ffcc04 3fffcc06
34:  ffffcc85 ffffcc05 ffffcc05 ffffffff
38:  ffffffff ffffffff ffffffff ffffffff

Exception:
3c:  33ffcc07 ffffffff ffffffff ffffffff
```

3.2 INITIAL ADS MEMORY MAP

The initial ADS memory map is the following:

Chip Select 0 - Flash Memory - 0x02800000 - 0x029fffff
Chip Select 1 - Board Control and Status register - 0x02100000 - 0x02103fff
Chip Select 2 - DRAM - 0x00000000 - 0x003fffff
Internal memory - 0x02200000

3.3 INITIALIZATION COMMAND FILE

When mpc8bug is activated, following reset and register initializations, it automatically executes the commands included in a configuration file. The configuration file is used to define aliases and memory mapped registers. It can also be used for other purposes, such as to reach a predefined state in a debug session.

The name of the automatic command file on a Sparcstation is: .mpc860.cfg
On a PC the name is mpc860.cfg

The EX command allows the user to execute other command files besides the default ones above.

CHAPTER 4
DEBUG MONITOR COMMANDS

4.1 ENTERING DEBUGGER COMMAND LINES

MPC8bug performs various operations in response to user commands entered at the keyboard. When the debugger prompt MPC860> appears on the terminal screen the debugger is ready to accept commands.

As the command line is entered it is stored in an internal buffer. Execution begins only after the carriage return is entered.

The debugger executes commands and returns the MPC860> prompt. However, if the entered command causes execution of user target code, (i.e., GO), then control may or may not return to the debugger. This depends upon the user program function. For example, if a breakpoint is specified, then control returns to the debugger when the breakpoint is encountered. Also refer to the paragraphs which detail elements of the GO commands.

In general debugger commands include:

A command identifier (i.e., MD or md for the memory display command). Both upper- or lower-case characters are allowed for command identifiers and options.

At least one intervening space before the first argument.

Any required arguments, as specified by command.

An option field, set off by a semicolon (:) to specify conditions other than the default conditions of the command.

Multiple debugger commands may be entered on a single command line by separating the commands with the explanation point (;) character.

The meta-symbols are:

<>	Angled brackets enclose a parameter value. The enclosed symbol may occur zero or one time. In some cases, where noted, characters in angled brackets are required.
[]	Square brackets enclose an optional symbol. The enclosed symbol may occur zero or one time.
[]…	Square brackets followed by periods enclose a symbol that is optional/repetitive. The symbol within the brackets may appear zero or more times.

|	This symbol indicates that a choice is to be made. Select one of several symbols separated by a straight line.
/	Select one or more of the symbols separated by the slash.
{ }	Brackets enclose optional symbols that may occur zero or more times.

4.1.1 User Interface

Command names, keywords, alias names, register names, data formats, and data constants are not case sensitive, while program symbols are case sensitive.

An address (addr) denotes an expression which consists of:

Operators consisting of: plus (+), minus (-), asterisk (*), slash (/), modulu (%), parenthesis (()), and period (.)

Constants (in hex/decimal/binary/char format).

General register names.

Special purpose registers: lr ctr iar

Symbol names – global variables and functions of the program.

Symbol name can be specified as they appear in the symbol table, or preceded by a $ sign.

A space is allowed between operators.

Example: .main+12+(r1*2)

When integer constants are expected as inputs, the default base is hexadecimal in all cases except:

In-line assembly where the default is decimal.

Anytime the constant's base is explicitly specified by a prefix:

0x - hex

0b - binary

& - decimal.

A constant can also be specified as a string: 'ccc' / "ccc"

When expecting an address, the debugger first tries to identify the input as a constant, then as register name, and then as a symbol. So symbol names, such as 'r1' 'a0', should be preceded by a '$' sign.

An address range is defined as start_range end_range.

Group names, register names, subfield names must be alphanumeric and start with a character.

Symbol and alias names can start with an alpha character (a-z or A-Z), period (.), underscore (_), slash (/), and tilde (~). While all characters after the first character can be alphanumeric characters (a-z, A-Z, and 0-9), period (.), underscore (_), slash (/), and tilde (~).

Symbol and alias name patterns can contain all previously defined characters as well as the wildcard characters: question marks (?) or asterisk (*).

File names can contain alphanumeric characters (a-z, A-Z, and 0-9), period (.), underscore (_), slash (/), tilde (~), caret (^), exclamation mark (!), at sign (@), percent (%), ampersand (&), plus (+), minus (-), equals (=), or the dollar sign ($) or colon (:).

The pattern is a string of alphanumeric characters. The string may include the wildcard characters: question marks (?) or asterisk (*). If the command contains a pattern value, the command cannot contain name or text values.

Many commands use <addr> as a parameter. All control addressing modes are allowed. An address+offset register mode is also allowed. The table below summarizes the address formats which are acceptable for address parameters in debugger command lines. Address shown below:

Debugger Address Parameter Format

Format	**Example**	**Description**
N	140	Absolute address.
N+rN	332+r5	Absolute address+contents of the specified offset register (not an assembler-accepted syntax).
Symbol Definition		
N - Absolute address (any valid expression) rN - General purpose register		

4.1.2 Syntactic Variables

The following syntactic variables are used in the command descriptions. In addition, other syntactic variables may be used and are defined in the particular command description in which they occur.

<DEL>	Delimiter; a space, or expression.
<ADDR>	Address or expression.
<COUNT>	A constant.

<RANGE>	A range of memory addresses which is specified by <ADDR><DEL> <ADDR>.
<TEXT>	An ASCII string of as many as 255 characters, delimited with single quote marks ('TEXT').

4.1.2.1 Expression as a Parameter

An expression is one or more numeric values separated by the arithmetic operators:

+	plus
–	minus
*	multiplied by
/	divided by
%	modulo
( )	

Base identifiers define numeric values as either a hexadecimal, decimal, octal or binary number.

Base	Identifier	Examples
Hexadecimal	0x	0xFFFFFFFF
Decimal	&	&1974, &10-&4
Binary	0b	0b1000110

If no base identifier is specified, then the numeric value is assumed to be hexadecimal.

A numeric value may also be expressed as a string literal of as many as four characters. The string literal must begin and end with single quote marks ('). The numeric value is interpreted as the concatenation of the ASCII values of the characters. This value is right-justified, as is any other numeric value.

String Literal	Numeric Value (in hex)
'A'	41
'ABC'	414243
'TEST'	54455354

Evaluation of an expression is always from left to right unless parentheses are used to group part of the expression. There is no operator precedence. Sub-expressions within parentheses are evaluated first. Nested parenthetical sub-expressions are evaluated from the inside out.

EXAMPLES Valid expressions.

Expression	Result (in Hex)
FF0011	FF0011
45+99	DE
&45+&99	90
0b10011110+0b1001	A7
88<<10	00880000

The total value of the expression must be between 0 and $FFFFFFFF.

4.1.2.2 Address as a Parameter

Many commands use <ADDR> as a parameter. An address+offset register mode is also allowed. Table 4-1 summarizes the address formats which are acceptable for address parameters in debugger command lines.

Table 4-1. Debugger Address Parameter Format

Format	Example	Description

N	140	Absolute address.
N+rN	332+r5	Absolute address+contents of the specified offset register (not an assembler-accepted syntax).
Symbol Definition		
N - Absolute address (any valid expression) rN - General purpose register		

4.2 DEBUGGER COMMANDS

Table 4-2 summarizes the MPCbug debugger commands.

Table 4-2. Debug Monitor Commands

Command & Syntax	Definition
A [<name> [<definition>]]	Display/define aliases
ARG [<addr><arg0> [<arg1...>]]	Set program line arguments
BR [[<opr>]<addr>|<start_range><end_range> [COUNT <n>]]	Set/Display instruction breakpoint
BRD [[<opr>]<addr>|<start_range><end_range>[READ|WRITE]] [COUNT <n>] [INSTRUCTION [[<opr>] <addr>|<start_range><end_range>]] [DATA [[<opr>]<data>|<data1 data2] [UNSIGNED] [:B|:H|:W]]	Set/Display data breakpoint
CAL <addr>	Calculate address expression
Ctrl-C	Abort
DEF <group><[.]reg_name> ["long_name"] <addr> [:W|:H|:B] DEF <group><[.]reg_name> BIT <subfield> ["long_name"] <mask> DEF [+|+csN] group_name [.] reg_name ["long_name"] addr [:W|H|B] [STATUS]	Define memory mapped register
EX <filename> [:Q]	Execute command file
GO [<addr>]	Execute user program
HELP|H [command]	Help

TABLE 4-2. Debug Monitor Commands (continued)

Command & Syntax	Definition
HIS [<num>]	Display commands history
LOAD <filename> [OFFSET <offset>] [NOBSS] [<heap_pointer>]	Load executable file from host
LOADF <filename><ram_addr>	Load executable file to flash memory
LOG [OFF | [WRITE][SHORT] <filename>]	Set log file
MBC <start_range><end_range><addr>	Memory block compare
MBF <start_range><end_range>,<data> [<increment>] [:B|:H|:W]	Memory block fill
MBM <start_range><end_range><addr>	Memory block move
MBS <start_range><end_range><data> [:B|H|W]	Memory block search
MD <start_range> [<end_range>] [:B|H|W|FS|FD|DB|DH|DW|I]	Memory display
MM <addr> [:B|H|W|C|FS|FD|I] [<data>]	Memory modify
NOA <name>	Delete alias
NOBR <id>|ALL	Breakpoint delete
PAGE	Enable/disable output paging
QUIT	Quit debugger
RD [<rN>] [:X|DB|DH|DW]	Register display
RDS [reg_name] RDS <group> [[.]reg_name]	Special purpose registers display
RESET	Reset board
RM [<rN> [<data>]]	Register modify

TABLE 4-2. Debug Monitor Commands (continued)

Command & Syntax	Definition
RMS [[<reg_name>] [[.]<subfield>] [<data>]] RMS <group> [[.][<reg_name>][[.]<subfield>]] [<data>]]	Special purpose register modify
SDEF <name><addr> SDEF [<pattern>] :RM	Attach/detach symbols
STDIN <filename> STDOUT <filename> STDERR <filename>	Redirect standard I/O to file
SYM [<pattern>]	Display symbol table
T [<num>] [<addr>] [:Q]	Trace user program
TC [<num>] [FROM <addr>] :Q	Trace on change of flow
UPM [:B] [entry_number value1 [value2...]	Display/Modify UPM setup
VE	Display debugger version number

A Display/Define Aliases

4.2.1 Display/Define Alias

```
A [<name> [<definition>]]
```

where:

<name>	String of alphanumeric characters which defines an alias.
<definition>	Expression of commands, operators, or operands.

The **A** command displays or defines aliases. The **A** command with no parameters displays currently defined aliases. The **A** command with a <name> parameter only displays that alias. When <name> and <definition> parameters are included the command defines a new alias. An alias can not exceed 256 characters. The maximum number of aliases is 100.

Note that the wildcard "*" can be used to shorten aliases. For example "rd*" can be used to display all aliases beginning with "rd".

EXAMPLES

```
MPCbug> a  mdl md $1 $1+$2-1<CR>
alias added
MPCbug>
```

```
MPCbug> mdl 1000 20<CR>
    ( md 1000 1000+20-1 )
00001000: 00000000 00000004 00000008 0000000c ................
00001010: 00000010 00000014 00000018 0000001c ................
MPCbug>
```

```
MPCbug> a rd_state rd r1; rd r0; rds ip<CR>
alias added
MPCbug>
```

A Display/Define Aliases

```
MPCbug> rd_state<CR>
        ( rd r1 )
   r1  ffffffff ....
        (  rd r0 )
   r0  ffffffff ....
       (  rds ip  )
    IP (SRR0) (Current Instruction Address)= 00001000


MPCbug> a<CR>
MDL          md $1 $1+$2-1
RD_STATE    rd r1; rd r0; rds ip
MPCbug>
```

ARG Set Program Line Arguments

4.2.2 Set Program Line Arguments

```
ARG [<addr><arg0> [<arg1...>]]
```

where:

<addr>	Address where the <arg0> parameter is to be located. If the command contains an <addr> value, the command also must contain an <arg0> value; the command may contain additional <arg> values (<arg1>, <arg2>, and so forth).
<arg0>	The program name argument. If the command contains an <arg0> value, it must also contain an <addr> value; the command may contain additional <arg> values (<arg1>, <arg2>, and so forth).
<arg1...>	Additional program arguments (<arg2>, <arg3>, and so forth).

The **ARG** command displays or sets program arguments. Entering the **ARG** command with no parameters displays the most recently set program arguments. Using <addr> and <arg> parameters loads the specified program arguments (<arg0><arg1>...) at the address (<addr>); setting r3 to argc, r4 to argv, and r5 (the UNIX environment variables pointer) to NULL.

ARG Set Program Line Arguments

EXAMPLE 1 Set program arguments to arg0="prog_name", arg1="arg1", arg2="arg2", arg3="aabbccdd" at address 0x2000:

```
MPCbug> arg 2000 prog_name arg1 arg2 aabbccdd<CR>
MPCbug>
```

EXAMPLE 2 Display program arguments

```
MPCbug> arg<CR>
program arguments were set to: prog_name arg1 arg2 aabbccdd
      at address=0x00002000
MPCbug>
```

EXAMPLE 3 Program argc and argv parameters, and memory contents after **ARG** command

```
MPCbug> rd r3; rd r4; rd r5<CR>
  r3  00000004 ....
  r4  00002020 ..
  r5  00000000 ....
MPCbug> md 2000<CR>
00002000: 70726f67 5f6e616d 65006172 67310061 prog_name.arg1.a
00002010: 72673200 61616262 63636464 00000000 rg2.aabbccdd....
00002020: 00002000 0000200a 0000200f 00002014 .. ... ... ... .
00002030: 00000000 00000000 00000000 00000000 ................
MPCbug>
```

BR **Instruction Breakpoint**

4.2.3 Instruction Breakpoint

```
BR [[<opr>]<addr>|<start_range><end_range> [COUNT <n>]]
```

where:

<opr>	An operator that sets a target code instruction address as a breakpoint address. A compare operator: = (equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), or != (not equal to). (For example, the command **BR < 4040** tells the monitor to set a breakpoint at the address range 0x0 thru 0x4040.)
<addr>	Address at which one breakpoint is to be set. If the command contains an <addr> value, it may also contain one compare operator, but not <start_range> and <end_range> values.
<start_range>	The first address of the range in which breakpoints are to be set. If the command contains an <start_range> value, it also must contain an <end_range> value, but not <opr> or <addr> values.
<end_range>	The last address of the range in which breakpoints are to be set. If the command contains an <end_range> value, it also must contain an <start_range> value, but not <opr> or <addr> values.
<n>	An integer, 1 through 0xFFFF (65,535): the number of times program execution passes through each breakpoint before stopping. The **BR** command may contain an <n> value only if the command includes the COUNT keyword.

The **BR** command displays or sets target code instruction breakpoints. The **BR** command with no parameters displays the current breakpoints in the breakpoint table. If an address or a range of addresses is specified with the **BR** command, that address is added to the breakpoint table. The <addr> and COUNT <n> parameters set a target code instruction address as a breakpoint address with count value <n>. If, during target code execution, a breakpoint with 0 count is found, the target code state is saved in the target registers and control returned to MPCbug. This allows the user to see the actual state of the processor at selected instructions in the code. As many as four instruction breakpoints can be defined.

BR Instruction Breakpoint

EXAMPLE 1 Set a breakpoint at all addresses greater than or equal to 1000, stop execution only after passing through the breakpoint 100 times

```
MPCbug>  br >= 1000 count 100<CR>
MPCbug>
```

EXAMPLE 2 Set a breakpoint at address proc1

```
MPCbug> br proc1<CR>
MPCbug>
```

EXAMPLE 3 Set a breakpoint through address range proc1 to proc1+0x20

```
MPCbug>  br proc1 proc1+20<CR>
MPCbug>
```

EXAMPLE 4 Display currently active instruction and data breakpoints

```
MPCbug> br<CR>
1.  >=0x00001000   cnt=256
        instruction breakpoint number 1    count=256
        address:  >=0x00001000
        registers used: CMPA COUNTA
2.  0x00002100
        instruction breakpoint number 2    count=1
        address:  0x00002100
                  ( proc1)
        registers used: CMPB
3.  0x00002100-0x00002120
        instruction breakpoint number 3    count=1
        address range:  0x00002100-0x00002120
                  ( proc1 - proc1+0x20)
        registers used: CMPC CMPD
MPCbug>
```

4.2.4 Data Breakpoint

```
BRD [[<opr>]<addr>|<start_range><end_range>[READ|WRITE]] [COUNT <n>]
[INSTRUCTION [[<opr>] <addr>|<start_range><end_range>]]
[DATA [[<opr>]<data>|<data1 data2] [UNSIGNED] [:B|:H|:W]]
```

where:

<opr>	A compare operator: = (equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), or != (not equal to). (For example, the command BRD < 4040 tells the monitor to set a data breakpoint at all the addresses less than 4040.)
<addr>	In the first phrase, the address at which one data breakpoint is to be set. In the INSTRUCTION phrase, the address of the specified store and load instruction. A compare operator may precede an <addr> value in either phrase. But if either phrase contains an <addr> value, the phrase can not contain <start_range> or <end_range> values.
<start_range>	In the first phrase, the first address of the range in which breakpoints are to be set. In the INSTRUCTION phrase, the address of the first instruction of the specified range. If either phrase contains an <start_range> value, it also must contain an <end_range> value, but not <opr> or <addr> values.
<end_range>	In the first phrase, the last address of the range in which breakpoints are to be set. In the INSTRUCTION phrase, the address of the last instruction of the specified range. If either phrase contains an <end_range> value, it also must contain a <start_range> value, but not <opr> or <addr> values.
<n>	An integer, 1 through 0xFFFF (65,535): the number of times program execution passes through each breakpoint before stopping. The **BRD** command may contain an <n> value only if the command includes the COUNT keyword.
<data>	The specified value of a DATA phrase. A compare operator may precede the <data> value. But if the phrase contains a <data> value, the phrase can not contain <data1> or <data2> values.
<data1>	The low value of a DATA-phrase range. If the phrase contains a <data1> value, it also must contain a <data2> value, but not opr or data values.
<data2>	The high value of a DATA-phrase range. If the phrase contains a <data2> value, it also must contain a <data1> value, but not <opr> or <addr> values.

BRD Data Breakpoint

The command syntax can be viewed as three optional phrases combined with a logical "and".

Address Phrase – Specifies the address range in memory of the written/read data and the READ|WRITE keyword specify if it on read (load) or write (store). (COUNT belongs to all specifications). The address phrase is:

```
[[<opr>]<addr>|<start_range><end_range>[READ|WRITE]] [COUNT <n>]
```

Instruction Phrase – Specifies the address range in memory of the store or load instruction, when executed, a breakpoint is activated. The instruction phrase is:

```
[INSTRUCTION [[<opr>] <addr>|<start_range><end_range>]]
```

Data Phrase – Specifies the range of <data> values stored or loaded to/from memory on which the data breakpoint should be set. The data phrase is:

```
[DATA [[<opr>]<data>|<data1><data2>] [UNSIGNED] [:B|:H|:W]]
```

The **BRD** command displays or sets target code data breakpoints. See below to reference **BRD** parameters

BRD Command Parameters	**Description**
BRD (no address, instruction, or data phrases)	Set a data breakpoint that will cause the program to stop on any load/store instruction.
BRD with address phrase (no instruction or data phrases)	A data breakpoint is set at the specified data address
BRD with address and instruction phrase (no data phrase)	Sets one data breakpoint, provided that the specified instruction or instructions caused the data to be at the specified address.
BRD with address and data phrase (no instruction phrase)	Sets one data breakpoint at the specified address, provided that the data conforms to the DATA-phrase criteria
BRD with address, instruction, and data phrases	Sets one data breakpoint at the specified address, provided that the specified instruction or instructions caused the data to be at that address, and also provided that the data conforms to the DATA-phrase criteria.

BRD Data Breakpoint

If you use the BRD command to set one breakpoint, you can include one of the compare operators, = through !=. For this type breakpoint, program execution stops at the first address that satisfies the compare-operator-and-address condition.

The first phrase of the **BRD** command may include either the READ or the WRITE keyword to specify read or write data breakpoints. If no READ or WRITE keyword is specified, this will be a read-or-write breakpoint.

The first phrase of this command may include the optional COUNT keyword and <n> parameter. If so, program execution passes through each breakpoint n times before stopping.

If you use this command to set a breakpoint, but there are insufficient free debug support registers for the specified breakpoint, the command does not set any breakpoints.

For the optional INSTRUCTION phrase to specify one instruction, use the instruction’s address as the <addr> value of the phrase; you may precede the <addr> value with a compare operator. For the INSTRUCTION phrase to specify several instructions; use the addresses of the first and last instructions as the <start_range> and <end_range> values of the phrase.

For the optional DATA phrase to specify one value, make it the <data> value of the phrase; you may precede the <data> value with a compare operator. For the DATA phrase to specify a value range, make the low and high range values the <data1> and <data2> values of the phrase. The DATA phrase may include the optional keyword UNSIGNED, and one of the optional values :B, :H, or :W (for 8-bit byte, 16-bit halfword, or 32-bit word, respectively), to further restrict the data specification.

BRD Data Breakpoint

EXAMPLE 1 Set breakpoint at the data store to memory at addresses range starting from ._vals to ._vals+0xFF (vals-a program array), stop execution only after passing through the breakpoint 10 times

```
MPCbug>  brd ._vals ._vals+ff write count 10<CR>
MPCbug>
```

EXAMPLE 2 Set breakpoint at the data store to memory of values 0x1000-0x1FFFFF in addresses range starts from ._vals to ._vals+0xFF, which performed from function _init (program defined function symbol) and stop execution only after passing through the breakpoint 10 times

```
MPCbug>  brd ._vals ._vals+ff write count 10 instruction _init _init_exit data 1000
1fffff<CR>
MPCbug>
```

CAL Calculate Address Expression

4.2.5 Calculate Address Expression

```
CAL <addr>
```

where:

<addr>	Address parameter to be calculated

Use the **CAL** command to calculate an address expression.

EXAMPLES

```
MPCbug> cal 2000 * 15 /3 + 778<CR>
the address: 0000e778
MPCbug>
```

```
MPCbug> cal r0+r1*2<CR>
the address: 00000aaa
MPCbug>
```

```
MPCbug> cal r0-ip + &32<CR>
the address: 00000142
MPCbug>
```

CTRL-C Abort User Program/Debugger Output

4.2.6 Abort User Program/Debugger Output

The **CTRL-C** key combination (pressing the CTRL and C keys simultaneously) aborts debugger output or user program execution on the Sun platform, by sending NMI to the board. On the IBM-PC the correct command is **CTRL-Break**. Unlike actual commands, the **CTRL-C/CTRL-Break** key combination has no command syntax. You can use this key combination to interrupt: MPCbug initialization, execution of the **GO**, **EX**, or **LOAD** commands and any other command that has a long output.

DEF Define Memory Mapped Register

4.2.7 Define Memory Mapped Register

```
DEF <group><[.]reg_name> ["long_name"] <addr> [:W|:H|:B]
DEF <group><[.]reg_name> BIT <subfield> ["long_name"] <mask>
DEF [+|+csN] group_name [.] reg_name ["long_name"] addr [:W|H|B] [STATUS]
```

where:

<group>	The name of the group that contains the special register.
<reg_name>	The name of the special register.
“long_name”	The optional long name describes the register or its subfield, in quotes.
<addr>	The address of the special register.
<subfield>	The name of the register subfield bits. Precede the <subfield> parameter with the keyword BIT.
<mask>	The register subfield bit mask.
+	means the register group address will be relative to the IMMR register value.
+csN	The register group address is an offset from chip select number N base address. (N=1-7). The + or +csN option can be specified with any register of the group. Both + and +csN options cannot be specified for two registers of the same group.
STATUS	Means that all the fields of the register are status bits and are cleared when 1 is written to them. When updating a single field in such a register, the debugger will take care to write 0 to all other fields.

The configuration file which includes the def commands will also include the mapping of BCSR board control and status register.

The **DEF** command defines a memory-mapped-special-purpose register or a subfield of a special purpose register.

In order to define a register, the command must include the register <group> name, <reg_name>, register <addr> and an optional register size (:W|:H|:B - the default register size is word).

In order to define register <subfield> bits, the command must include the register <group> name, <reg_name> (must be previously defined) and <subfield> bit name preceded by the BIT keyword and bit <mask> value.

The command may include a string "long_name" consisting of a full register name or any other user defined string.

DIAG Start Diagnostic Monitor

4.2.8 Start Diagnostic Monitor

```
DIAG
```

Use the **DIAG** command to enter the diagnostic monitor. Once you are in the diagnostic monitor you can enter **HELP** for a list of the available commands. Enter **EXIT** to exit the diagnostic monitor. See Chapter 6 for more information.

EXAMPLE 1 To enter the diagnostic monitor:

```
MPCbug> diag<CR>
MPCdiag%>
```

EXAMPLE 2 To exit the diagnostic monitor:

```
MPCdiag%> exit<CR>
MPCbug>
```

EX Execute Debugger Command File

4.2.9 Execute Command File

```
EX <filename> [:Q]
```

where:

<filename>	The name of a command script file.

The **EX** command executes a series of ADS commands from the file defined as <filename>. Executing such a file has the same effect as executing the individual commands in sequence. A command file is convenient for any sequence of commands often needed; the command file saves time and promotes accuracy.

The syntax shows that the **EX** command can include the optional :Q quiet mode indicator, which directs MPCbug to execute the commands of the file without displaying the commands or their outputs.

Note that the **EX** command itself can be in a command file. In this way, you can nest command files as many as 16 levels deep.

The following features are available on the Sun version only: When the up-arrow button is pressed, debugger commands may be edited from the history. Also the "Tab" button may be used to complete filenames, as well as ctrl-d may be used to get a list of files.

Debugger startup includes execution of the command script file .mpc860.cfg (or mpc860.cfg on PC) This file should be in the current or home directory.

EXAMPLES

MPCbug> **ex init.scr<CR>** Execute commands in file INIT.SCR (.SCR extension is optional).

GO Execute User Program

4.2.10 Execute User Program

```
GO [<addr>]
```

where:

<addr>	The starting address for program execution.

Use the **GO** command to initiate target code execution. All breakpoints that have been previously set are still enabled. If an address is specified, execution starts at that address, it is placed in the target instruction pointer (IP). Execution starts at the target IP address.

EXAMPLE 1 Begin program execution at IP:

```
MPCbug> go<CR>
```

EXAMPLE 2 Begin program execution at address 0x4000:

```
MPCbug> go 4000<CR>
```

HELP Help

4.2.11 Help

```
HELP|H [<command>]
```

where:

<command>	Display command description for a specific command acronym.

The **HELP** (or **H**) command activates the MPCbug help. If you enter this command without a command parameter value, the complete list of MPCbug commands appears; select **H** <command> and the corresponding help information appears.

HELP Help

EXAMPLE 1

```
MPCbug> H<CR>
syntax: H|HELP [command]
  HELP is available on the following commands:
A                         display/define alias
ARG                       set program line arguments
BR                        set/display instruction breakpoint
BRD                       set/display data breakpoint
CAL                       calculate address expression
DEF                       define memory mapped register
DIAG                      board diagnostic command
EX                        execute debugger command file
GO                        execute user program
HELP/H                    help
HIS                       display commands history
LOAD                      load user program
LOADF                     load user program to flash memory
LOG                       set log file
MBC                       memory block compare
MBF                       memory block fill
MBM                       memory block move
MBS                       memory block search
MD                        memory display
MM                        memory modify
NOA                       delete alias
NOBR                      breakpoint delete
PAGE                      toggle output paging on/off
QUIT                      quit debugger
RD                        register display
RDS                       special purpose/memory mapped registers display
RESET                     send reset to board
RM                        register modify
RMS                       special purpose register modify
SDEF                      attach/detach symbols
STDIN/STDOUT/STDERR       redirect standard I/O to file
SYM                       display symbol table
T                         trace user program
TC                        trace on change of flow
UPM                       display/modify UPM RAM
VE                        print debugger version
MPCbug>
```

EXAMPLE 2 Display help information on the command RESET

```
MPCbug> H VE<CR>
syntax:  VE
      print debugger version number.
MPCbug>
```

HIS Display Commands History

4.2.12 Display Commands History

```
HIS [<num>]
```

where:

<num>	The number of commands to be displayed

Use the **HIS** command to display the command history list. If the <num> parameter is specified, only the latest commands are displayed notto exceed the <num> parameter value. The **HIS** command can display, at most, 200 of most recent commands. If an **EX** command is one of the commands that has been executed, only the **EX** command is displayed, not the commands contained in the script file.

EXAMPLE

```
MPCbug> his<CR>
1: load xcoff.exec
2: rm r1 1ff00
3: br 4000
4: go
5: his
MPCbug>
```

LOAD Load Executable File From Host

4.2.13 Load Executable From Host

```
LOAD <filename> [OFFSET <offset>] [NOBSS] [<heap_pointer>]
```

where:

<filename>	Name of the file to be loaded.
<offset>	Optional parameter preceded by the OFFSET keyword specifies an offset value. When an <offset> is specified you may load S-record format modules shifted by the offset value.
<heap_pointer>	Heap pointer address value may be specified by the user. If <heap_pointer> is not specified, it is set by default at end of the loaded sections.

Use the LOAD command to download an executable file from a host computer to the ADS. The LOAD command accepts data from the host and loads it into on-board memory. Types of executable files accepted by the ADS are:

XCOFF

COFF

ELF (and little endian ELF format)

S-Records

Loading XCOFF, COFF, ELF might modify the r2 register (TOC pointer) and the IP register, clears r3 and r5 registers and loads the programs symbol table (after removing any previously defined symbols).

The S-record format (refer to Appendix A) allows a specified entry point in the address field of the S-record-block termination record. The contents of the termination-record address field (plus any offset address) is put into the target IP. Thus, after a download, enter **GO** without an <addr> parameter to execute the downloaded code. An error condition exists if the embedded-record checksum does not agree with the checksum calculated by MPCbug. A checksum error is a fatal error and causes the **LOAD** command to abort. (NOTE: in later versions the checksum error is a warning only).

If the NOBSS keyword is specified the BSS section is not cleared when **LOAD** is executed and if no <heap_pointer> parameter is specified, it is set by default at the end of the loaded section.

LOAD Load Executable File From Host

EXAMPLE 1 Load XCOFF program simple.xcoff to target memory

```
MPCbug> load  simple.xcoff<CR>
Loading XCOFF file . . .
Loading section 2 (.text) : 000000a0 bytes at 00000000
Loading section 4 (.data) : 0000002c bytes at 00001000
Loading section 5 (.bss) : 00000010 bytes at 0000102c (not loaded)
Entry point (IP) set to 00000000
r2 (toc) set to 00001020
Heap start address set to 0000103c
Loaded 00000011 symbols into the symbol table
Duplicated symbols (2) expanded with enumerated suffixes
r3 and r5 are set to 0
MPCbug>
```

EXAMPLE 2 Load S-records program simple.mx to target memory

```
MPCbug>  simple.mx<CR>
Loading Srecords file . . .
Loading block : at 00001000
              : 000000c4 bytes loaded
Loading block : at 00002000
              : 0000004c bytes loaded
Entry point (IP) is not set
Heap start address set to 0000204c
r3 and r5 are set to 0
MPCbug>
```

LOADF Load Executable File To Flash Memory

4.2.14 Load Executable File To Flash Memory

```
LOADF <filename><ram_addr>
```

where:

<filename>	Name of the file to be loaded.
<ram_addr>	Buffer address in on-board memory.

Use the **LOADF** command to download an executable file from the host computer to the ADS flash memory. The **LOADF** command accepts a file from the host and loads that data contained in it into flash on-board memory to the address specified in the host file, using a temporary buffer at <ram_addr> specified on **LOADF** command line. The size of the buffer in on-board memory should equal the executable file size plus ~0x4000 bytes. Types of files accepted by the ADS are:

XCOFF

COFF

ELF

S-Records

Loading XCOFF, COFF, and ELF modifies the r2 register (TOC pointer) and the IP register, clears r3 and r5 registers and loads the programs symbol table (after removing any previously defined symbols). Also, when using the **LOADF** command to load XCOFF, COFF, and ELF format files, only the flash memory mapped sections of code are loaded into the flash memory, all other sections are loaded to on-board memory defined by its address.

LOADF Load Executable File To Flash Memory

EXAMPLE Load the XCOFF program *simple2.xcoff* to flash memory

```
MPCbug> loadf simple_flash.xcoff 3000<CR>
loadf: Loading XCOFF file . . .
Loading flash mapped sections to ram memory buffer:
Loading section 2 (.text) : 000001a0 bytes at 00003000
Programming flash : 000001a0 bytes at 02800000-0280019f
Flash programming completed

Loading ram mapped sections to ram memory:
Loading section 4 (.data) : 00000060 bytes at 00010000
Loading section 5 (.bss) : 00000018 bytes at 00010060 (not loaded)
r2 (toc) set to 00010048
Entry point (IP) set to 01000000
Heap start address set to 00010078
Loaded 00000018 symbols into the symbol table
Duplicated symbols (2) expanded with enumerated suffixes
MPCbug>
```

LOG Start/Stop Log To File

4.2.15 Start/Stop Log To File

```
LOG [OFF | [WRITE][SHORT] <filename>]
```

where:

<filename>	Name of the file used to store log data.

Use the LOG command to record debugger commands and all command outputs to a specified file. Entering LOG with no parameter values prints the current log status. The OFF keyword toggles the logging function “ON” and “OFF”. The WRITE keyword overwrites previous file contents; otherwise MPCbug appends log outputs to the file. The SHORT keyword logs only commands, not outputs, during the current session.

EXAMPLE Log debugger commands to out.log file

```
MPCbug> log write out.log<CR>
log is write to out.log
MPCbug>
```

4.2.16 Memory Block Compare

MBC <start_range><end_range><addr>

where:

<start_range>	The starting address for the range (block) of memory to be compared.
<end_range>	The last address for the range (block) of memory to be compared.
<addr>	The first address for the second range or block of memory.

The MBC command compares the contents of two specified memory ranges, or blocks, displaying mismatch notifications. If the <start_range> address is greater than the <end_range> address, an error results.

EXAMPLE 1 Memory compare, nothing printed

```
MPCbug> mbc 0 ff 3300<CR>
  compare 00000000-000000ff   to 00003300
MPCbug>
```

EXAMPLE 2 Create a mismatch

```
MPCbug> mm 50:b<CR>
00000050= aa  ? 57.
MPCbug>
```

EXAMPLE 3 Mismatches are printed out

```
MPCbug> mbc 0 ff 3300<CR>
  compare 00000000-000000ff   to 00003300
00000050: 57bb0000               00003350: aabb0000
MPCbug>
```

4.2.17 Memory Block Fill

```
MBF <start_range><end_range>,<data> [<increment>] [:B|:H|:W]
```

where:

<start_range>	The starting address for the range or block of memory to be filled.
<end_range>	The last address for the range or block of memory to be filled.
<data>	Constant value.
<increment>	Increment value for each successive write.

The MBF command fills the specified range of memory with a data pattern. If the command does not include an increment value, each address of the range receives the data value. If the command does include an increment value, each successive address of the range receives an incremented data value. (A negative increment value creates a decremented pattern).

If the <end_range> is not a correct boundary for an integer multiple of the <data>, MPCbug fills <data> to the last boundary before the <end_range>. Effective address messages show the extent of the area that receives <data> values.

The <data> values are right-justified. An optional value :B, :H, or :W specifies the data size as an 8-bit byte, 16-bit halfword, or 32-bit word, respectively. (The default is :W.). If a data value does not fit into the specified field, MPCbug prints an error message.

MBF Memory Block Fill

EXAMPLE 1 Fill memory range 0x1000-0x10ff with data 0xAABB0000

```
MPCbug> mbf 1000 10ff, aabb0000<CR>
  fill aabb0000  in 00001000-000010ff
MPCbug>
```

EXAMPLE 2 Display the filled memory

```
MPCbug> md 1000<CR>
00001000: aabb0000 aabb0000 aabb0000 aabb0000 ................
00001010: aabb0000 aabb0000 aabb0000 aabb0000 ................
00001020: aabb0000 aabb0000 aabb0000 aabb0000 ................
00001030: aabb0000 aabb0000 aabb0000 aabb0000 ................
MPCbug>
```

EXAMPLE 3 Fill memory range 0x1000-0x10ff with data 0xAA00 and increment each halfword by 2

```
MPCbug> mbf 1000 10ff, aa00 2 :h<CR>
  fill aa00  in 00001000-000010ff
MPCbug>
```

EXAMPLE 4 Display the filled memory

```
MPCbug> md 1000<CR>
00001000: aa00aa02 aa04aa06 aa08aa0a aa0caa0e ................
00001010: aa10aa12 aa14aa16 aa18aa1a aa1caa1e ................
00001020: aa20aa22 aa24aa26 aa28aa2a aa2caa2e . .".$.&.(.*.,..
00001030: aa30aa32 aa34aa36 aa38aa3a aa3caa3e .0.2.4.6.8.:.<.>
MPCbug>
```

MBM Memory Block Move

3.4.18 Memory Block Move

```
MBM <start_range><end_range><addr>
```

where:

<start_range>	The starting address for the range or block of memory to be moved.
<end_range>	The last address for the range or block of memory to be moved.
<addr>	The first address of the new memory location for the data.

The MBM command copies the contents of a memory block determined by <start_range> and <end_range> to another place in memory defined by <addr>.

EXAMPLES

```
MPCbug> md 1000<CR>
00001000: 54686973 20697320 61207465 73742021 This is a test !
00001010: 21210000 00000000 00000000 00000000 !!..............
00001020: 00000000 00000000 00000000 00000000 ................
00001030: 00000000 00000000 00000000 00000000 ................
MPCbug>
```

```
MPCbug> mbm 1000 103f 4000<CR>
  move from 00001000-0000103f   to 00004000
MPCbug>
```

```
MPCbug> md 4000<CR>
00004000: 54686973 20697320 61207465 73742021 This is a test !
00004010: 21210000 00000000 00000000 00000000 !!..............
00004020: 00000000 00000000 00000000 00000000 ................
00004030: 00000000 00000000 00000000 00000000 ................
MPCbug>
```

4.2.19 Memory Block Search

```
MBS <start_range><end_range><data> [:B|H|W]
```

where:

<start_range>	The starting address for the range or block of memory to be searched.
<end_range>	The last address for the range or block of memory to be searched.
<data>	The object of the search.

The **MBS** command searches the specified range of memory for a match with a user entered data pattern.

In this mode, a data pattern is entered as a part of the command line.

Use the size option field to set the compare alignment as successive bytes, halfwords or words (the default is word) within the range, for a match with your defined data. The data value is right-justified, MPCbug truncates leading bits or adds leading zeros to make the data pattern the specified size. If a match is found, then the address of the first byte of the match is displayed. If the [:b|h|w] parameter is not specified, there will be noalignment during the search (i.e. 0x12345678 will be found in: 00000000: 00123456 78000000 ....)."

MBS Memory Block Search

EXAMPLE 1 Assuming you run the **MD** command and the following data is in memory at address 1000. Then when the search is initiated and the string is found, the address of the first matched byte is displayed.

```
MPCbug> md 1000<CR>
00001000: 54657374 20526573 756c7473 203a0000 Test Results :..
00001010: 00000000 00000000 00000000 00000000 ................
00001020: 00000000 00000000 00000000 00000000 ................
00001030: 00000000 00000000 00000000 00000000 ................
MPCbug> mbs 500 1fff, 'Test'<CR>
 unaligned search 54657374  in 00000500-00001fff
00001000
MPCbug>
```

EXAMPLE 2 Assuming you run the **MD** command and the following data is in memory at address 2000. Then when the data search mode is initiated and the string is found, the address of the first matched byte is displayed

```
MPCbug> md 2000<CR>
00002000: 00000000 12345678 00000000 00000000 .....4Vx........
00002010: 00000000 00000000 00000000 00000000 ................
00002020: 00000000 00000000 00000000 00000000 ................
00002030: 00000000 00000000 00000000 00000000 ................
MPCbug> mbs 1500 2500, 12345678<CR>
 unaligned search 12345678  in 00001500-00002500
00002004
MPCbug>
```

4.2.20 Memory Display

```
MD <start_range> [<end_range>] [:B|H|W|FS|FD|DB|DH|DW|I]
```

where:

<start_range>	The starting address for the range or block of memory to be displayed.
<end_range>	The last address for the range or block of memory to be displayed.

MD accepts the following data formats:

B	Byte
H	Halfword
W	Word
FS	Floating point single
FD	Floating point double
DB	Decimal byte
DH	Decimal half word
DW	Decimal word
I	Instruction

Use the **MD** command to display the contents of multiple memory locations. The default data type is word. When format is not specified the **MD** command displays data in hex and character format. If <end_range> is not given the **MD** command displays a 64 byte block. If the command includes optional formatting requirements ([:B|H|W|FS|FD|DB|DH|DW]), the memory is displayed in that format. The instruction (I) format displays disassembled memory assembly code.

MD Memory Display

EXAMPLES

```
MPCbug> md 3000<CR>
00003000: aabb0000 aabb0004 aabb0008 aabb000c ................
00003010: aabb0010 aabb0014 aabb0018 aabb001c ................
00003020: aabb0020 aabb0024 aabb0028 aabb002c ... ...$...(...,
00003030: aabb0030 aabb0034 aabb0038 aabb003c ...0...4...8...<
MPCbug>
```

```
MPCbug> md 3000 305f :fs<CR>
00003000:  -3.321787e-13  -3.321788e-13  -3.321789e-13  -3.321791e-13
00003010:  -3.321792e-13  -3.321793e-13  -3.321794e-13  -3.321795e-13
00003020:  -3.321796e-13  -3.321797e-13  -3.321798e-13  -3.321799e-13
00003030:  -3.321800e-13  -3.321801e-13  -3.321802e-13  -3.321804e-13
00003040:  -3.321805e-13  -3.321806e-13  -3.321807e-13  -3.321808e-13
00003050:  -3.321809e-13  -3.321810e-13  -3.321811e-13  -3.321812e-13
MPCbug>
```

```
MPCbug> md 3000 305f :fd<CR>
00003000: -7.5343435537e-103 -7.5343776174e-103
00003010: -7.5344116811e-103 -7.5344457448e-103
00003020: -7.5344798084e-103 -7.5345138721e-103
00003030: -7.5345479358e-103 -7.5345819995e-103
00003040: -7.5346160631e-103 -7.5346501268e-103
00003050: -7.5346841905e-103 -7.5347182542e-103
MPCbug>
```

```
MPCbug> md 3000 305f :dh<CR>
00003000: -21829     0 -21829     4 -21829     8 -21829    12
00003010: -21829    16 -21829    20 -21829    24 -21829    28
00003020: -21829    32 -21829    36 -21829    40 -21829    44
00003030: -21829    48 -21829    52 -21829    56 -21829    60
00003040: -21829    64 -21829    68 -21829    72 -21829    76
00003050: -21829    80 -21829    84 -21829    88 -21829    92
MPCbug>
```

```
MPCbug> md 3000 305f :dw<CR>
00003000: -1430585344 -1430585340 -1430585336 -1430585332
00003010: -1430585328 -1430585324 -1430585320 -1430585316
00003020: -1430585312 -1430585308 -1430585304 -1430585300
00003030: -1430585296 -1430585292 -1430585288 -1430585284
00003040: -1430585280 -1430585276 -1430585272 -1430585268
00003050: -1430585264 -1430585260 -1430585256 -1430585252
MPCbug>
```

MD Memory Display

```
MPCbug> **md 0 :i<CR>**
0x00000000:          82420000 lwz    r18, 0x0(r2)
0x00000004:          80f20008 lwz    r7, 0x8(r18)
0x00000008:          39000000 addi   r8,r0, 0x0
0x0000000c:          91070000 stw    r8, 0x0(r7)
0x00000010:          81320000 lwz    r9, 0x0(r18)
0x00000014:          90690000 stw    r3, 0x0(r9)
0x00000018:          81320004 lwz    r9, 0x4(r18)
0x0000001c:          90890000 stw    r4, 0x0(r9)
0x00000020:          80e20004 lwz    r7, 0x4(r2)
0x00000024:          91070000 stw    r8, 0x0(r7)
0x00000028:          48000051 bl     0x00000078
0x0000002c:          4ffffb82 cror   31,31,31
0x00000030:          80f20008 lwz    r7, 0x8(r18)
0x00000034:          80e70000 lwz    r7, 0x0(r7)
0x00000038:          2c070000 cmpi   0,r7, 0x0
0x0000003c:          4182000c bc     0xc,2,0x00000048
MPCbug>
```

4.2.21 Memory Modify

```
MM <addr> [:B|H|W|C|FS|FD|I] [<data>]
```

where:

<addr>	The address of a memory location.
<data>	Data to store at memory location defined by addr.

MM accepts the following data formats:

B	Byte
H	Halfword
W	Word
C	Character String
FS	Floating point single
FD	Floating point double
I	Instruction

Use the **MM** command to examine and change the contents of a memory location. When a <data> value is specified the memory address is loaded with that value. If no <data> is specified, interactive memory modify is initiated.

The default data type is word. The **MM** command reads and displays the contents of memory at the specified address and prompts the user with a question mark (?). The user may enter new data for the memory location, followed by <**CR**>, or may simply enter <**CR**>, which leaves the contents unaltered. That memory location is closed and the next memory location is opened.

When the character [:C] format is specified the debugger accepts a variable character string length. If the command includes optional formatting requirements ([:B|H|W|C|FS|FD|I]), the memory is modified in that format.

MM Memory Modify

In all formats except [:C] and [:I] the user may enter one of several step control characters, either at the prompt or after writing new data. Enter one of the following step control characters to modify the command execution:

V or v	The next successive memory location is opened. This is the default. It initializes whenever MM is executed and remains initialized until changed by entering one of the other special characters.
^	MM backs up and opens the previous memory location.
=	MM re-opens the same memory location. This is useful for examining I/O registers or memory locations that are changing over time).
.	Terminates MM command. Control returns to MPCbug.

The I format activates the one-line assembler/disassembler. The control characters are disabled if :I or :C formats are selected.

EXAMPLE 1 Access location 4000, modify three words and backup for check

```
MPCbug> mm 4000<CR>
00004000= 54686973  ? 1<CR>
00004004= 20697320  ? 2<CR>
00004008= 61207465  ? 3<CR>
0000400c= 73742021  ? ^<CR>
00004008= 00000003  ?
00004004= 00000002  ?
00004000= 00000001  ? .<CR>
MPCbug>
```

EXAMPLE 2 Access location 2000, modify two floating point double words

```
MPCbug> mm 2000 :fd<CR>
00002000=   1.5089747817e-315   ? 12345.6789<CR>
00002008=    0.0000000000e+00   ? 12.00<CR>
00002010=    0.0000000000e+00   ? .<CR>
MPCbug>
```

MM Memory Modify

EXAMPLE 3 Enter a new source instructions

```
MPCbug>  mm  100  :i<CR>
0x00000100=    00000000 unknown primary opcode 0 ? addi r1,r1,0x1<CR>
0x00000104=    00000000 unknown primary opcode 0 ? b 0x100 <CR>
0x00000108=    00000000 unknown primary opcode 0 ? .<CR>
MPCbug>  md  100  107  :  i<CR>
0x00000100:           38210001 addi   r1,r1, 0x1
0x00000104:           4bfffffc b      0x00000100
```

NOA Delete Alias

4.2.22 Breakpoint Delete

```
NOA <name>
```

where:

<name>	String of alphanumeric characters: the alias name.

Delete alias with specified name.

EXAMPLE Define and delete an alias

```
MPCbug> a mdl md $1 $1+$2-1<CR>
alias added
MPCbug>

MPCbug> noa mdl<CR>
alias deleted
MPCbug>
```

NOBR Breakpoint Delete

4.2.23 Breakpoint Delete

```
NOBR <id>|ALL
```

where:

<id>	Breakpoint id in the debugger breakpoint table (can be seen by BR command).

Use the **NOBR** command to delete breakpoints from the breakpoint table. The **NOBR** command deletes both instruction and data breakpoints from the breakpoint table. To remove a specific breakpoint from the breakpoint table, enter **NOBR** followed by the breakpoint identification number (<id>). Using the **NOBR** command with the keyword ALL deletes all entries from the breakpoint table.

EXAMPLE 1 Display all current breakpoint id's

```
MPCbug> br<CR>
1.  >=0x00001000   cnt=256
        instruction breakpoint number 1    count=256
        address:  >=0x00001000
        registers used: CMPA COUNTA
2.  0x00002100
        instruction breakpoint number 2    count=1
        address:  0x00002100
                  ( proc1)
        registers used: CMPB
3.  0x00002100-0x00002120
        instruction breakpoint number 3    count=1
        address range:  0x00002100-0x00002120
                  ( proc1 - proc1+0x20)
        registers used: CMPC CMPD
```

EXAMPLE 2 Delete breakpoint id=1

```
MPCbug>  nobr 1<CR>
breakpoint deleted
MPCbug>
```

4.2.24 Enable/Disable Output Paging

```
PAGE
```

Use the **PAGE** command to toggle paging ON/OFF. When ON, paging lets you display 24 lines of data on the screen. Entering a carriage return (<**CR**>) lets you view the next 24 lines of data.

QUIT Terminate Debugger

4.2.25 Terminate Debugger

```
QUIT
```

The QUIT command terminates the debugger.

EXAMPLE

```
MPCbug> QUIT<CR>              Terminates the debugger
%>
```

4.2.26 Display Registers

```
RD [<rN>] [:X|DB|DH|DW]
```

where:

<rN>	General purpose register r0-r31.

The **RD** command accepts the following data formats:

X	hex
DB	Decimal byte
DH	Decimal half word
DW	Decimal word

Use the RD command to display all general purpose registers. If rN is specified the command displays the specified register only. When the RD command is executed without options the display is in hex format followed by a character format.

EXAMPLE 1

```
MPCbug> rd<CR>
  r0-r3   00000000 00000001 00000002 00000003   ................
  r4-r7   00000004 00000005 00000006 00000007   ................
 r8-r11   00000008 00000009 0000000a 0000000b   ................
r12-r15   0000000c 0000000d 0000000e 0000000f   ................
r16-r19   00000000 00000000 00000000 00000000   ................
r20-r23   00000000 00000000 00000000 00000000   ................
r24-r27   00000000 00000000 00000000 00000000   ................
r28-r31   ffffffff ffffffff 00000000 00000000   ................
MPCbug>
```

EXAMPLE 2

```
MPCbug> rd r28 :dw<CR>
  r28          -1
MPCbug>
```

RDS Display Special Purpose Registers

4.2.27 Display Special Purpose Register

RDS [reg_name]
RDS <group> [[.]reg_name]

where:

<reg_name>	Special purpose register to be displayed with it's subfield.
<group_name>	Name of the user defined register group to be displayed.

Use the **RDS** command to display special purpose registers and list user defined special purpose register groups. Optional arguments let you display any core register or user defined register groups.

EXAMPLE

```
MPCbug> RDS <CR>
MPCbug>
```

RESET Reset Board

4.2.28 Reset Board

RESET [:H] [:NI] [:50|20]

<:H>	Perform hard reset instead of soft reset.
<:NI>	Do not perform any initializations after the reset. Note that correct operation is not guaranteed after this operation.
<:50|20>	Perform hard reset and initialize the memory controller and UPM for board frequency of 50 MHz/20 MHz. (NOTE: this option may not be available in your current release).

Use the **RESET** command to reset the ADS. Each time a reset command is activated, it resets the device and execute the sequence of settings described in the Chapter 2.

EXAMPLE

```
MPCbug> RESET<CR>
MPCbug>
```

RM Modify Register

4.2.29 Modify Register

```
RM [<rN> [<data>]]
```

where:

<rN>	General purpose register r0-r31.
<data>	Data to be written to the register defined by rN.

Use the **RM** command to display and change the contents of general purpose registers. When <rN> and <data> values are specified, the general purpose register is loaded with that value. If no data is specified, interactive register modification mode is initiated.

When in interactive mode, if an optional register is specified, the interactive mode starts from that register, otherwise it starts from r0. The **RM** command reads and displays the contents of a general purpose register and prompts the user with a question mark (?). The user may enter new data for the displayed register, followed by <**CR**>, or simply enter <**CR**>. Entering a <**CR**> without a new value leaves the register contents unaltered, it is closed and the next general purpose register is opened. A period (**.**) followed by a <**CR**> terminates **RM** command.

This command accepts the same control characters as the **MM** command.

EXAMPLES

```
MPCbug> rm r4<CR>
r4=0x00000000 ? 1<CR>
r5=0x00000000 ? 2<CR>
r6=0x00000000 ? 3<CR>
r7=0x00000810 ? 4<CR>
r8=0x00000000 ? 5<CR>
r9=0x00000850 ? .<CR>
MPCbug>
```

```
MPCbug> rm r10 6<CR>
MPCbug>
```

RMS Modify Special Purpose Register

4.2.30 Modify Special Purpose Register

```
RMS [[<reg_name>] [[.]<subfield>] [<data>]]
RMS <group> [[.][<reg_name>][[.]<subfield>]] [<data>]]
```

where:

<reg_name>	The name of the special purpose register.
<subfield>	The name of a subfield within a special purpose register.
<data>	Data to be written to the special purpose register or its subfield.
<group>	The name of the group of defined special purpose registers.

Use the **RMS** command to display and change the contents of special purpose registers. When a special purpose register and data value are specified, the special purpose register is loaded with that value. If no data is specified, interactive register modification mode is initiated.

When in interactive mode, if an optional register is specified, the interactive mode starts from that register, otherwise it starts from the instruction pointer (IP). The **RMS** command reads and displays the contents of a special purpose register and prompts the user with a question mark (?). The user may enter new data for the displayed register, followed by <**CR**>, or simply enter <**CR**>. Entering a <**CR**> without a new value leaves the register contents unaltered, it is closed and the next general special purpose is opened. A period (**.**) followed by a <**CR**> terminates **RMS** command.

This command accepts the same control characters as the **MM** command.

If a <group> parameter is specified, **RMS** displays and changes the contents of user group defined special purpose registers. Then the command functions in the same way as described in the above paragraphs for core special purpose registers.

SDEF Assign/Remove Symbols

4.2.31 Assign/Remove Symbols

```
SDEF <name><addr>
SDEF [<pattern>] :RM
```

where:

<name>	Symbol name can only contain alphanumeric characters.
<addr>	Symbol address, the value which substitutes symbol name upon address calculation.
<pattern>	Can contain alphanumeric and wildcard characters.

Use the **SDEF** command to assign or remove symbols to or from the symbol table. In order to assign a symbol to the symbol table, <name> and <addr> parameters must be specified. Use the **SDEF** command and :RM keyword to remove a particular symbol from the symbol table.

EXAMPLE 1 Display the symbol table of the program simple.xcoff (LOAD example)

```
MPCbug> sym<CR>
                name   address
            .__start = 0x00000000
             __start = 0x00001018
              _adata = 0x00001000
                 TOC = 0x00001020
            _adata_1 = 0x00001020
             errno_1 = 0x00001024
             p_xargc = 0x0000102c
             p_xargv = 0x00001030
            p_xrcfg = 0x00001034
               p_xrc = 0x00001038
           syscall.c = 0x00001010
               errno = 0x00001010
             environ = 0x00001014
               .main = 0x00000078
               .exit = 0x0000007c
               .exit = 0x0000007c
                exit = 0x00001028

MPCbug>
```

SDEF Attach/Detach Symbols

EXAMPLE 2 Assign a new symbol to the symbol table

```
MPCbug>  sdef  my_exxit  .exit+10<CR>
symbol my_exxit = 0x0000008c  added to symbol table
MPCbug>
```

EXAMPLE 3 Remove all other "exit" symbols from symbol table and display the symbol table.

```
MPCbug>  sdef  *exit*  :rm<CR>
                  name   address
                 .exit = 0x0000007c  removed from symbol table
                 .exit = 0x0000007c  removed from symbol table
                  exit = 0x00001028  removed from symbol table

MPCbug>  sym<CR>
                  name   address
              .__start = 0x00000000
               __start = 0x00001018
                _adata = 0x00001000
                   TOC = 0x00001020
              _adata_1 = 0x00001020
               errno_1 = 0x00001024
               p_xargc = 0x0000102c
               p_xargv = 0x00001030
              p_xrcfg = 0x00001034
                 p_xrc = 0x00001038
             syscall.c = 0x00001010
                 errno = 0x00001010
               environ = 0x00001014
                 .main = 0x00000078
              my_exxit = 0x0000008c

MPCbug>
```

STDIN — Standard Input From a File
STDOUT — Standard Output to a File
STDERR — Standard Error Output to a File

4.2.32 Redirect Standard I/O

```
STDIN <filename>
STDOUT <filename>
STDERR <filename>
```

where:

<filename>	The name of the file the standard I/O be read from or written to.

These three commands redirect a program’s standard I/O to or from a file. The default standard I/O is the host computer keyboard and screen.

SYM Print Symbols

4.2.33 Print Symbols

```
SYM [<pattern>]
```

where:

<pattern>	Can contain alphanumeric and wildcard characters.

Use the **SYM** command to display the attached symbol table, search the attached symbol table for a particular symbol pattern and search the attached symbol table for a set of symbols from symbol table.

EXAMPLE Symbol table of the program simple.xcoff (LOAD example)

```
MPCbug> sym<CR>
                      name    address
                  .__start = 0x00000000
                   __start = 0x00001018
                    _adata = 0x00001000
                       TOC = 0x00001020
                  _adata_1 = 0x00001020
                   errno_1 = 0x00001024
                   p_xargc = 0x0000102c
                   p_xargv = 0x00001030
                   p_xrcfg = 0x00001034
                     p_xrc = 0x00001038
                 syscall.c = 0x00001010
                     errno = 0x00001010
                   environ = 0x00001014
                     .main = 0x00000078
                     .exit = 0x0000007c
                     .exit = 0x0000007c
                      exit = 0x00001028

MPCbug>
```

4.2.34 Trace

```
T [<num>] [<addr>] [:Q]
```

where:

<num>	Specifies the number of instructions to be traced.
<addr>	Starting address for the trace command

Use the **T** command to execute one instruction at a time and display the next instruction to be executed. **T** starts tracing at the address in the target IP or from the given address field <addr> if FROM keyword is specified . The optional <num> parameter specifies the number of instructions to be traced before returning control to MPCbug. The <num> parameter default is 1. As each instruction is traced, the next instruction is disassembled and displayed.

When tracing, breakpoints are monitored (but not inserted) for all trace commands. In all cases, if a breakpoint with 0 count is encountered, control is returned to MPCbug.

TRACE is implemented with the trace bit SE in MSR. Do not modify trace bit SE while using the trace commands. Code in ROM or FLASH can be traced, because the trace functions are implemented using the hardware trace bits in the MPC860 device. During trace mode, breakpoints are monitored and their counts decremented when the corresponding instruction with breakpoint is traced.

The **TRACE** command can include the optional :Q quiet mode indicator, which directs MPCbug to execute the trace command without displaying the on the debugger screen the executed instructions.

T   Trace

EXAMPLE 1 Tracing on the first 10 instructions of program simple.xcoff (LOAD example)

```
MPCbug>  md 0 3f :i<CR>
.__start:            82420000 lwz     r18, 0x0(r2)
.__start+0x4:        80f20008 lwz     r7, 0x8(r18)
.__start+0x8:        39000000 addi    r8,r0, 0x0
.__start+0xc:        91070000 stw     r8, 0x0(r7)
.__start+0x10:       81320000 lwz     r9, 0x0(r18)
.__start+0x14:       90690000 stw     r3, 0x0(r9)
.__start+0x18:       81320004 lwz     r9, 0x4(r18)
.__start+0x1c:       90890000 stw     r4, 0x0(r9)
.__start+0x20:       80e20004 lwz     r7, 0x4(r2)
.__start+0x24:       91070000 stw     r8, 0x0(r7)
.__start+0x28:       48000051 bl      .main
.__start+0x2c:       4ffffb82 cror    31,31,31
.__start+0x30:       80f20008 lwz     r7, 0x8(r18)
.__start+0x34:       80e70000 lwz     r7, 0x0(r7)
.__start+0x38:       2c070000 cmpi    0,r7, 0x0
.__start+0x3c:       4182000c bc      0xc,2,.__start+0x48
MPCbug>
```

EXAMPLE 2 Trace 12 instructions

```
MPCbug>  t &12<CR>
.__start+0x4     0x00000004    80f20008 lwz     r7, 0x8(r18)
.__start+0x8     0x00000008    39000000 addi    r8,r0, 0x0
.__start+0xc     0x0000000c    91070000 stw     r8, 0x0(r7)
.__start+0x10    0x00000010    81320000 lwz     r9, 0x0(r18)
.__start+0x14    0x00000014    90690000 stw     r3, 0x0(r9)
.__start+0x18    0x00000018    81320004 lwz     r9, 0x4(r18)
.__start+0x1c    0x0000001c    90890000 stw     r4, 0x0(r9)
.__start+0x20    0x00000020    80e20004 lwz     r7, 0x4(r2)
.__start+0x24    0x00000024    91070000 stw     r8, 0x0(r7)
.__start+0x28    0x00000028    48000051 bl      .main
.main            0x00000078    48000000 b       .main
.main            0x00000078    48000000 b       .main
MPCbug>
```

EXAMPLE 3 Trace 11 instructions in quite mode

```
MPCbug>  t &11 :q<CR>
MPCbug>  t<CR>
.main            0x00000078    48000000 b       .main
MPCbug>
```

TC **Trace on Change of Control Flow**

4.2.35 Trace on Change of Control Flow

```
TC [<num>] [FROM <addr>] :Q
```

where:

<num>	Specifies the number of instructions to be traced.
<addr>	Starting address of the TC single step command.

Use the **TC** command to trace on detection of an instruction that causes a change of control flow, such as B, BC, BCCTR, etc. and display the target next command to be executed. Execution is in real time until a change of flow instruction is encountered. The <num> parameter specifies the number of change of flow instructions to be traced before returning control to MPCbug. The <num> parameter default is 1. **TC** starts tracing at the address in the target IP or from a given <addr> parameter if the FROM keyword is specified . As each instruction is traced, a next instruction is disassembled and its display printout generated.

During tracing, breakpoints are monitored (but not inserted) for all trace commands. In all cases, if a breakpoint with 0 count is encountered, control is returned to MPCbug. Note that the **TC** command recognizes a breakpoint only if it is at a change of flow instruction.

TC implemented with the trace bit BE in MSR register. Do not modify trace bit BE while using the trace commands. Code in ROM or FLASH can be traced, because the trace functions are implemented using the hardware trace bits in the MPC860 device. During trace mode, breakpoints are monitored and their counts decremented when the corresponding instruction with breakpoint is traced.

The **TC** command can include the optional :Q quiet mode indicator, which directs MPCbug to execute the **TC** command without displaying the executed instructions on the debugger screen.

TC Single Step

EXAMPLE **TC** on the of program simple.xcoff (T example)

```
MPCbug> tc<CR>
.main             0x00000078    48000000 b       .main
MPCbug>
```

UPM DISPLAY/MODIFY UPM SETUP

4.2.36 Display/Modify UPM Setup

```
UPM [:B] [entry_number1 [entry_number2]]
and
UPM [:B] entry_number, value1 [value2 ...]
```

where:

<:B>	Specifies the UPM, either A or B.
<entry_number>	Specifies the UPM entry number. If two entry numbers are specified, that range is displayed.
<value1 etc.>	Specifies the values that the UPM should be written with, beginning with the specified entry number.

EXAMPLE 1 **Display UPM B Contents**

MPCbug> **UPM B** <**CR**>

EXAMPLE 2 **Display UPM B Entry 3**

MPCbug> **UPM B 3** <**CR**>

EXAMPLE 3 **Display UPM B Entry-Range 3 to 5**

MPCbug> **UPM B 3 5** <**CR**>

EXAMPLE 4 **Modify UPM B Starting with Entry 3**

MPCbug> **UPM B 3,** 00ffec00 37ffec47 ffffffff ffffffff <**CR**>

VE DISPLAY VERSION NUMBER

4.2.37 Display Debugger Version Number

<u>**EXAMPLE**</u> **Display Debugger Version Number**

MPCbug> **VE** <**CR**>

CHAPTER 5
DEBUGGING AN EXCEPTION HANDLER

5.1 DEBUGGING EXCEPTIONS

To cause a user written software exception handler routine to gain control when an exception occurs, the matching DER (Debug Enable Register) bit must be cleared. For example, to reach the decrementer exception handler enter: rms der decie 0

It is not allowed to set breakpoints or to single-step inside an exception handler's epilog or prologue. (I.E. before srr0,srr1 are saved or after they are restored.) Before setting a breakpoint in an exception handler, the BRKNOMSK in the LCTRL2 register must be set, otherwise the breakpoint will have no effect (i.e. do: rms lctrl2 brknomsk 1 ; br ...).

CHAPTER 6
DIAGNOSTIC MODE

6.1 DIAG COMMAND

The diagnostic mode tests are called via a command line driven diagnostic monitor.

DIAG

At the debugger prompt, the DIAG command switches the debugger to diagnostic mode. The prompt should now read MPCdiag%>.

The following menu is displayed upon entry to diag mode:

```
--------------------------------------------------
Diagnostic Mode Commands
--------------------------------------------------
  ST - Self test.  Runs tests T1-T5
  T1 - ADI host - board communications test
  T2 - Chip reset and initialization test
  T3 - RAM march test  [0x00000000-0x003fffff]
  T4 - RAM walk bit test  [0x00000000-0x003fffff]
  T5 - Flash memory march test  [0x02800000-0x029fffff]
  T6 - UART port test
  T7 - Ethernet port test

Other tests and commands

  EXIT   exit diagnostic mode
  HELP   diagnostic mode help
  DMB    diagnostic memory bounds
  HTL    host transmit loop
  RL     memory read loop
  WL     memory write loop
```

6.2 DIAGNOSTIC MODE COMMANDS DESCRIPTION

T1

Opens and tests the host - board connection in loopback mode by first writing 0-255 to every byte, and then alternatively writing 0x55, 0xbb 1000 times.

T2

Hard resets the board and does the register initializations.

T3

Tests the board's RAM. Performs 3 march tests (0x55555555 0xaaaaaaaa and address) from start to the stop memory address.

T4

Tests the SRAM. Performs 64 walking bit tests from start to the stop memory address. The first 32 walking bit test performed with 32-bit value with only one bit set, and the other 32 tests performed with the complement of 32-bit value with only one bit set.

T5

Tests the Flash memory. Performs 3 march tests (0x55555555 0xaaaaaaaa and address) from start to the stop memory address. Each sector before being erased is saved in SRAM and restored after the test.

T6

Uart port test. Performs a transmit/receive from/to the Uart port. This test requires a loopback connector on the UART port.

T7

Ethernet port test. Performs a transmit/receive from/to the Ethernet port. This test requires a loopback connector on the Ethernet port.

DMB <start_addr> <stop_addr>

Diagnostic Memory Bounds. Allows a user to select start and stop address used by the RAM tests. Stop address should be set higher than 0xa00.

HPT [tty_device]

Host Peripheral Test. The serial device parameter [sdev] is optional, the default is the serial device the debugger set to. This test opens the serial device, and checks the transmit/receive of the whole characters set through the port. This tests demands a loopback connector between TxD and RxD pins of the serial port.

HTL

Host Transmit Loop. Two characters 0x55, 0xbb are alternatingly send in an infinite loop from the host via the board's ADI in a loopback mode. Ctrl-C (or Ctrl-Break on the PC) terminates HTL.

RL address

Read Loop. RL executes a streamlined read from the specified address. Ctrl-C (or Ctrl-Break on the PC) terminates the RL.

WL address value1 [value2]

Write Loop. WR executes a streamlined intermittently write to the specified address of value1 and value2. If value2 is not specified then value2 is taken as complement of value1. Ctrl-C (or Ctrl-Break on the PC) terminates the command.

CHAPTER 7
IN-LINE ASSEMBLER

7.1 DESCRIPTION

An instruction consists of a mnemonic (+possible extensions) and a list of operands separated by commas. Each operand must be a number. By default it is recognized as decimal. If preceded by 0x it is recognized as hexadecimal. If preceded by 0b it is recognized as binary. If it is bracketed it is recognized as a character literal ('ABC'). (An n-byte character literal is translated to an 8n bit field).

- rn/Rn 0<=n<=31 are recognized as the number n.
- Space tab ',' '#' '(' ')' are all recognized as delimiters between instruction parts.
- Everything after # is considered to be a comment.

The assembler does not evaluate expressions. The assembler first checks if the mnemonic is valid. Then the number of operands and their sizes are compared to those associated with the mnemonic. In the mtspr,mfspr instructions both symbolic SPR names and decimal SPR numbers are accepted.

Branch addresses:
- Can be written as an absolute number or as $+offset, $-offset, where $ specifies the location counter. * can be used instead of the $.
- Also can be written as: symbol, symbol+offset, symbol-offset.
- Independently of the above, the branch address will be kept in opcode as an absolute number (shifted right 2 bits) if "absolute" bit is on, or as an offset from the location counter otherwise.

Extended mnemonics are currently not implemented. Assembly code produced by the debugger’s disassembler is guaranteed to pass in-line assembly. Branch addresses in disassembly are represented as symbol+offset if a symbol exists for which the offset is less than 0x10000.