VM LABS

Puffin: A Debugger For NUON™

Revision 1.30
5-May-99

Note: This document is continually updated to reflect the current state of the Nuon development system hardware and software. If you have a version that is more than five or six months old, it is likely out of date.

Please address comments or report errors to Mike Fulton at VM Labs (mfulton@vmlabs.com).

VM Labs, Inc.
520 San Antonio Road
Mountain View, CA 94040

Tel: (650) 917-8050
Fax: (650) 917-8052

Table of Contents

1. INTRODUCTION 1-1

1.1 Usage 1-1
1.2 Environment Variables 1-1
1.2.1 MD_PORT 1-1
1.2.2 MD_LOGFILE 1-1
1.2.3 PUFFIN_PATH 1-2
1.3 User Customization Files 1-2
1.3.1 USER.LSP 1-2
1.3.2 USER2K.LSP 1-2
1.4 Global Variables 1-3
1.4.1 MMP and &m 1-3
*1.4.2 *MPE& and &p 1-3*
1.4.3 &p0, &p1, &p2, and &p3 1-3
1.5 Customizable Global Variables 1-3
*1.5.1 *STEP-OVER-INTERRUPTS* 1-3*
*1.5.2 *DETECT-CONFLICTS* 1-4*
*1.5.3 *DISPLAY-WARNINGS* 1-4*
*1.5.4 *DISPLAY-INFO* 1-4*

2. DEBUGGING FUNCTIONS 2-1

2.1 Select A Processor 2-1
2.1.1 `select-processor` 2-1
2.2 File Loading Functions 2-1
2.2.1 `load-debug-file` 2-1
2.2.2 `load-source-file` 2-1
2.2.3 `load-and-run-source-file` 2-2
2.2.4 `load-object-file` 2-2
2.2.5 `load-and-run-object-file` 2-3
2.2.6 `load-symbols` 2-3
2.2.7 `load-binary-file` 2-3
2.2.8 `set-source-path` 2-4
2.3 Execution Control Functions 2-4
2.3.1 `run` 2-4
2.3.2 `step` 2-4
2.3.3 `step-over` 2-4
2.3.4 `stop` 2-4
2.3.5 `restart` 2-4
2.4 Breakpoint Functions 2-5

2.4.1 setbp *2-5*
2.4.2 clearbp *2-5*
2.4.3 showbp *2-5*
2.5 Watch Functions 2-5
2.5.1 watch symbol *2-5*
2.5.2 watch-change *2-6*
2.5.3 unwatch *2-6*
2.5.4 define-bitfield *2-6*
2.6 Before/After Methods 2-7
2.6.1 before *2-7*
2.6.2 remove-before *2-7*
2.6.3 after *2-7*
2.6.4 remove-after *2-8*
2.7 Image Output Functions 2-8
2.7.1 write-image *2-8*
2.7.2 write-raw-image *2-8*
2.8 Miscellaneous Functions 2-8
2.8.1 disassemble *2-8*
2.8.2 dump *2-9*
2.8.3 runtime-eval *2-9*
2.8.4 find-symbol *2-9*

3. THE DEBUGGING ARCHITECTURE 3-1

4. THE CHIP ABSTRACTION 4-1

4.1 C-MMP Class 4-1
4.1.1 Global Variables 4-1
4.1.2 Methods 4-1
4.2 C-MPE Class 4-4
4.2.1 Global Variables 4-4
4.2.2 Methods 4-4
4.2.3 Execution Control Methods 4-8
4.2.4 Breakpoint Methods 4-9
4.2.5 Object File Access Methods 4-11
4.3 C-REGISTER Class 4-14
4.3.1 Methods 4-14
4.4 C-BREAKPOINT Class 4-14
4.4.1 Methods 4-15

5. UTILITY FUNCTIONS 5-1

5.1 Fixed Point Math Operations 5-1

6. DELETED INFORMATION FROM HERE ON! 6-1

6.1.1 Emulator-Only Methods 6-1

6.1.2 Other misc methods ... 6-3
6.1.3 Emulator Only Methods ... 6-5
6.2 BASIC-VECTOR-REGISTER Class ... 6-6
6.2.1 Methods ... 6-6
6.3 VECTOR-REGISTER Class ... 6-6
6.3.1 Methods ... 6-7
6.4 VECTOR-IO-REGISTER Class ... 6-7
6.4.1 Methods ... 6-7
6.5 ACCUMULATOR Class ... 6-7
6.5.1 Emulator Only Methods ... 6-7

7. THE DEBUGGER ABSTRACTION ... 7-1

7.1 DEBUGGER Class ... 7-1
7.1.1 Global Variables ... 7-1
7.1.2 Methods ... 7-1
7.2 MPE-DEBUGGER Class ... 7-2
7.2.1 Global Variables ... 7-2
7.2.2 Methods ... 7-2
7.2.3 Emulator Only Methods ... 7-5

This page intentionally left blank.

1. Introduction

Puffin is a programmable debugger for the Nuon Architecture. It supports the standard debugger features of breakpoints and single stepping along with powerful features like before and after methods. Puffin uses a scripting language called XLISP which is similar to the programming language Scheme.

1.1 Usage

There are two versions of Puffin, one with a GUI interface and one with a command line interface. The GUI version of Puffin is called **Puffin2K**, The command line version is called simply **Puffin**. When you invoke either version from the command line, you can supply optional arguments. These arguments should be the names of files you want loaded. The files should contain debugger scripts (XLISP code).

1.2 Environment Variables

Puffin uses some environment variables to control its execution. You should set these variables before invoking Puffin.

1.2.1 MD_PORT

If you are debugging a Nuon hardware system, you should set the environment variable MD_PORT to the IP address of the Nuon hardware system. For example:

```
set MD_PORT=192.1.1.226
```

This variable is also used by other tools in the Nuon SDK.

1.2.2 MD_LOGFILE

This environment variable is mostly used to debug Puffin itself. Set it to the name of a file if you want logging information about TCP/IP requests and responses sent to the Nuon hardware written into the file.

For example:

```
set MD_LOGFILE=mpacket.log
```

Note: Do not set this variable unless you really need the resulting log file. Writing to the log can slow down debugger operations and in particular downloads significantly.

1.2.3 PUFFIN_PATH

It isn't necessary to set this variable if you have installed the Puffin debugger along with the rest of the Nuon SDK and are using the standard paths relative to the directory specified by the **VMLABS** environment variable.

If you want to place the XLISP files (PUFFIN.LSP, MMP.LSP, MPE.LSP, etc.) in different directories from those used by the standard distribution, set this environment variable to a path where these files can be found. For example:

```
set PUFFIN_PATH=d:\work\lisp;d:\work\other
```

This directs Puffin to search for XLISP files in the specified directories in the order in which they are listed.

1.3 User Customization Files

You can customize the behavior of Puffin using one of the user customization files described below. These files are automatically loaded by Puffin during startup and can override the default settings of Puffin global variables.

1.3.1 USER.LSP

The USER.LSP file is loaded by both Puffin and Puffin2K. You should place in USER.LSP any customizations that you want to apply to both the command line and GUI versions of Puffin.

1.3.2 USER2K.LSP

The USER2K.LSP file is loaded only by Puffin2K. You should place in USER2K.LSP any customizations that you want to apply only to Puffin2k.

1.4 Global Variables

1.4.1 MMP and &m

The variable *MMP* is always bound to an instance of the C-MMP class. The variable *&m* is a shorthand for *MMP*.

1.4.2 *MPE& and &p

The variable *MPE* is always bound to the instance of C-MPE corresponding to the currently selected MPE. The variable *&p* is a shorthand for *MPE*. Within a before or after method or within the conditional expression of a conditional breakpoint, these variables are temporarily rebound to the instance of C-MPE associated with the MPE that has reached the associated breakpoint.

1.4.3 &p0, &p1, &p2, and &p3

The variables *&p0*, *&p1*, *&p2* and *&p3* are bound to the corresponding MPEs.

1.5 Customizable Global Variables

Puffin uses some global variables within its scripting language to control the debugger operation. You can override the values of any of these variables by placing the appropriate commands in your USER.LSP or GUI-USER.LSP files or in a debugger script files. You can also temporarily override a variable by typing the appropriate command at the command prompt in Puffin or in the listener window in Puffin2K.

1.5.1 *STEP-OVER-INTERRUPTS*

When *step-over-interrupts* is set to **#t** (which it is by default), Puffin attempts to step over the execution of the interrupt service routine of any interrupt that occurs during single stepping. To disable stepping over interrupts, set this variable to #f.

To enable stepping over interrupts:

```
(set! *step-over-interrupts* #t)
```

To disable stepping over interrupts:

```
(set! *step-over-interrupts* #f)
```

1.5.2 *DETECT-CONFLICTS*

When *detect-conflicts* is set to **#t** (which it is by default), Puffin halts and displays an error message when it detects a conflict within an instruction packet. Set this variable to **#f** to ignore conflicts.

1.5.3 *DISPLAY-WARNINGS*

When *display-warnings* is set to **#t** (which it is by default), Puffin will display warnings produced by LLAMA (which it uses to assemble source files during the load process). To suppress the display of warnings, set this variable to **#f**.

1.5.4 *DISPLAY-INFO*

When *display-info* is set to **#t** (which it is by default), Puffin will display informational messages produced by LLAMA (which it uses to assemble source files during the load process). To suppress the display of informational messages, set this variable to **#f**.

2. Debugging Functions

2.1 Select A Processor

2.1.1 select-processor

```
 (select-processor i)
```

Selects the specified processor in the specified debugger. This changes the binding of the variables *mpe* and *&p* to the new MPE.

2.2 File Loading Functions

2.2.1 load-debug-file

```
(load-debug-file filename)
```

Load a file containing XLISP code. This file usually contains commands to initialize a debugging session including selecting processors and loading source or object files. It can also be used to setup watch variables and breakpoints as well as to define functions needed for the current debugging session.

2.2.2 load-source-file

```
(load-source-file filename &key processor ignore-
before-after? use-fast-loader? load-debugging-info?
load-code? initialize? run?)
```

Load Nuon source code into the mpe associated with the debugger.

:processor selects the target mpe. The default is &p.

:ignore-before-after? should be set to #t to ignore before and after methods. Its default value is #f.

:use-fast-loader? should be set to #t to use the fast loader. The fast loader uses a helper program that it loads into the mpe being loaded. It only really speeds up loading into SDRAM or system ram. If you are only loading on-chip memory, it is probably faster to set this parameter to #f. The default is #t.

:load-debugging-info? should be set to #t to load debugging information (symbols and line numbers). You can set it to #f if you just want to load the code and don't care about debugging. The default is #t.

:load-code? should be set to #t to load code and data. You can set it to #f if you just want to load debugging information. The default is #t.

:initialize? should be set to #t to initialize the program for debugging. When this parameter is set to #t, the debugger sets a breakpoint at the first instruction of the program and runs until it hits the breakpoint. This has the effect of loading the instruction pipeline and setting *pcexec* to the start address of the program. If :initialize? is set to #f, the program is loaded and its start address is placed in *pcfetch*. The default is #t.

:run? should be set to #t to automatically start running after loading the object file. If :run? is set to #f, the object file is loaded but not started. The default is #f.

Note: This function used to be called mload.

2.2.3 load-and-run-source-file

```
(load-source-file filename &key processor ignore-
before-after? use-fast-loader? load-debugging-info?
load-code?)
```

Assemble, load, and start running Nuon source code. This function does the same thing as the **load-source-file** function with the **:run?** parameter set to #t.

2.2.4 load-object-file

```
(load-object-file filename &key processor ignore-
before-after? use-fast-loader? load-debugging-info?
load-code? initialize? run?)
```

Load Nuon object file (either a .cof file or an .mpo file) into the mpe associated with the debugger.

:processor selects the target mpe. The default is &p.

:ignore-before-after? should be set to #t to ignore before and after methods. Its default value is #f.

:use-fast-loader? should be set to #t to use the fast loader. The fast loader uses a helper program that it loads into the MPE being loaded. It only really speeds up loading into SDRAM or system ram. If you are only loading on-chip memory, it is probably faster to set this parameter to #f. The default is #t.

:load-debugging-info? should be set to #t to load debugging information (symbols and line numbers). You can set it to #f if you just want to load the code and don't care about debugging. The default is #t.

:load-code? should be set to #t to load code and data. You can set it to #f if you just want to load debugging information. The default is #t.

:initialize? should be set to #t to initialize the program for debugging. When this parameter is set to #t, the debugger sets a breakpoint at the first instruction of the program and runs until it hits the breakpoint. This has the effect of loading the instruction pipeline and setting *pcexec* to the start address of the program. If :initialize? is set to #f, the program is loaded and its start address is placed in *pcfetch*. The default is #t.

:run? should be set to #t to automatically start running after loading the object file. If :run? is set to #f, the object file is loaded but not started. The default is #f.

2.2.5 load-and-run-object-file

```
(load-object-file filename &key processor ignore-
before-after? use-fast-loader? load-debugging-info? )
```

Load and start running a Nuon object file. This function does the same thing as the **load-object-file** function with the **:run?** parameter set to #t

2.2.6 load-symbols

```
(load-symbols filename &key processor)
```

Load the symbols and line numbers from a Nuon object file for use with the specified MPE.

: processor selects the target MPE. The default is &p.

2.2.7 load-binary-file

```
(load-binary-file addr filename)
```

Load a binary file at the specified address MPE associated with the debugger.

Note: This function used to be called **bload**.

2.2.8 set-source-path

```
(set-source-path! path &optional debugger)
```

Sets the path the debugger uses to find source files. The path should be in the form of a list of strings naming directories where files are to be found. The directory separator should be the forward slash (/) even on Windows 95/98 systems. The debugger defaults to &d.

2.3 Execution Control Functions

2.3.1 run

```
(run &optional processor)
```

Start the MPE running. The processor defaults to &d.

2.3.2 step

```
(step &optional processor)
```

Single step the MPE. The processor defaults to &d.

2.3.3 step-over

```
(step-over &optional processor)
```

Single step the MPE, stepping over subroutine calls. The processor defaults to &d.

2.3.4 stop

```
(stop &optional processor)
```

Stop the MPE. The processor defaults to &d.

2.3.5 restart

```
(restart)
```

Re-initializes the MMP and reloads the last program loaded into an MPE or the last debugger file loaded.

2.4 Breakpoint Functions

2.4.1 setbp

```
(setbp addr &key condition)
```

Set a breakpoint at the specified addresses in the currently selected processor. If the condition keyword parameter is specified, it should be an expression that returns true (any value other than #f) if execution should stop and #f if it should continue without stopping. Puffin will evaluate the expression when the breakpoint is encountered and will stop if the expression evaluates to true. If the expression evaluates to #f, Puffin will automatically continue without stopping.

Within the expression, the names of registers are bound to their associated register objects. In addition, the symbol &p is bound to the MPE object.

For example:

```
(setbp "loop" :condition '(< (r1 'value) 2)))
```

This will break at the label "loop" if the value of r1 is less than 2.

2.4.2 clearbp

```
(clearbp addr…)
```

Clear breakpoints at the specified addresses in the currently selected processor.

2.4.3 showbp

```
(showbp)
```

Show all active breakpoints in the currently selected processor.

2.5 Watch Functions

2.5.1 watch symbol

```
(watch symbol &key format popup-format fracbits count
local? use-cache? indirect?)
```

Setup to watch the specified variable. Returns an id for the watch request.

Watch keyword parameters:

:format	one of 'hex, 'binary, 'decimal, 'real or 'ascii (default is 'hex)
:popup-format	the name of a bitfield format defined with define-bitfield (no default)
:fracbits	number of bits in the fractional part of a 'real or 'binary value (default is 0)
:count	number of elements in an array of values (default is 1)
:local?	use the mpe memory map if #t and the global memory map if #f (default is #t)
:use-cache?	look through the data cache if #t (default is #f)
:indirect?	treat the value as a pointer and display the value pointed to (default is #f)

2.5.2 watch-change

```
(watch-change id &key format popup-format fracbits
count local? use-cache? indirect?)
```

Changes the settings of an existing watch request. The keywords are the same as in the watch function above.

2.5.3 unwatch

```
(unwatch id)
```

Remove the watch request with the specified id.

2.5.4 define-bitfield

```
(define-bitfield name &rest fields)
```

Define a named bitfield definition to be used with the :popup-format parameter to the watch function. Each field is a string of the form:

"<label>.<start-bit>:<end-bit>.<format>"

where <label> is a the label that will appear to the left of the field value in the popup, <start-bit> is the bit number of the start of the field, <end> is the bit number of the end of the field and <format> is a printf style format string for displaying the field. For example:

```
(define-bitfield "my_bitfield"
      "this.0:1.%d"
      "that.2:3.%x"
      "other.4:31.%d")
```

This defines a bitfield definition called "my_bitfield" that consists of three field definitons. The first has the label "this" and starts with bit 0 and ends with bit 1. It is displayed with the format %d which converts the value to decimal. The second field has the label "that" and starts with bit 2 and ends with bit 3. It is displayed with the format %x which converts the value to hexadecimal.

2.6 Before/After Methods

2.6.1 before

```
(before addr &rest body)
```

Establishes a before method at the specified address. When execution reaches the specified address, the code in the body of the before method is executed before the instruction at that address is executed.

For example:

```
(before #x80001000
  (if (< (r0 'value) 0)
    (format #t "~%R0 has gone negative!")))
```

This will establish a before method at the address $80001000 that tests the value of r0 and displays a message in the console window if the value of r0 is less than zero.

Within the expression, the names of registers are bound to their associated register objects. In addition, the symbol &p is bound to the MPE object.

2.6.2 remove-before

```
(remove-before addr)
```

Remove the before method at the specified address.

2.6.3 after

```
(after addr &rest body)
```

Establishes an after method at the specified address. When execution reaches the specified address, the code in the body of the after method is executed after the instruction at that address is executed. This function is analogous to the before function above except that the code is executed after the instruction at the specified address.

2.6.4 remove-after

```
(remove-after addr)
```

Remove the after method at the specified address.

2.7 Image Output Functions

2.7.1 write-image

```
(write-image name &optional x-size y-size &key base
mode mpe)
```

Write an image from display memory to a .PCX bitmap image file.

The x-size and y-size parameters default to the display height and width. The base defaults to the start of external ram and the mode defaults to *display-mode*.

2.7.2 write-raw-image

```
(write-raw-image name &optional x-size y-size &key
base mode mpe)
```

Write an image from display memory to a .PCX bitmap image file.

The x-size and y-size parameters default to the display height and width. The base defaults to the start of external ram and the mode defaults to *display-mode*.

This function differs from **write-image** in that no color space conversion is performed; the Y component of colors is written into the green channel of the output image, Cr into the red, and Cb into the blue.

2.8 Miscellaneous Functions

2.8.1 disassemble

```
(disassemble addr count &key port processor)
```

Disassemble instructions in the selected processor starting at the specified address. The port defaults to *standard-output*. The processor defaults to &p.

2.8.2 dump

```
(dump &optional processor)
```

Dump the registers of the specified MPE. The processor defaults to &p.

2.8.3 runtime-eval

```
(runtime-eval expr &optional processor)
```

Evaluate the specified expression in the processor context. This include bindings for *mpe* and &p as well as for each processor register (e.g. r0, r1, mdmacptr, etc.).

2.8.4 find-symbol

```
(find-symbol pname &optional processor)
```

Find the value of the named symbol. The processor defaults to &p

See also the definition of the **find-symbol** method of C-MPE.

This page intentionally left blank.

3. The Debugging Architecture

Puffin supports debugging Nuon programs by providing an abstraction of the Nuon processor. This abstraction is presented in the form of XLISP classes. This set of classes is separated into two major categories:

- The Chip Abstraction
- The Debugger Abstraction

This page intentionally left blank.

4. The Chip Abstraction

The chip abstraction consists of the classes MMP and MPE.

4.1 C-MMP Class

The C-MMP class is an abstraction of the entire Nuon chip. Puffin only supports a single instance of the C-MMP class.

4.1.1 Global Variables

- ***mmp*** is always set to the only instance of C-MMP
- **&m** is a synonym for *mmp*.

4.1.2 Methods

4.1.2.1 mpe-count

```
(c-mmp ‘mpe-count)
```

Returns the number of MPEs associated with this C-MMP. For example:

```
(&m ‘mpe-count)
  ➔  4
```

4.1.2.2 mpe

```
(c-mmp ‘mpe i)
```

Returns the *i*th MPE associated with this C-MMP. For example:

(&m ‘mpe 2)

➔ #<MPE-2>

4.1.2.3 fetch-scalar

```
(c-mmp 'fetch-scalar addr)
```

Returns the scalar at the specified address in MMP memory. MPE memory appears as with a DMA transfer with the REMOTE bit set.

4.1.2.4 store-scalar

```
 (c-mmp 'store-scalar! addr value)
```

Stores the specified value into the specified address in MMP memory. MPE memory appears as with a DMA transfer with the REMOTE bit set.

4.1.2.5 read-scalars-from-file

```
(c-mmp 'read-scalars-from-file addr count
 &optional port)
```

Reads scalars from the specified input port and writes them to the specified address. The data should be in binary form in Nuon (big-endian) byte order.

4.1.2.6 write-scalars-to-file

```
(c-mmp 'write-scalars-to-file addr count
 &optional port)
```

Reads scalars from the specified address and writes them to the specified output port. The data will be in binary form in Nuon (big-endian) byte order.

4.1.2.7 chip

```
(c-mmp 'start)
```

Returns #t if running on a real chip. It returns #f when running under the emulator (which is not present in this release).

4.1.2.8 run-all

```
(c-mmp 'run-all)
```

Start all MPEs running. Each MPE will execute instructions when the MMP is clocked. When debugging on actual hardware, the MMP is always being clocked.

4.1.2.9 stop-all

```
(c-mmp 'stop-all)
```

Stop all MPEs from running.

4.1.2.10 reset

```
(c-mmp 'reset &key keep-breakpoints?)
```

Reset the MMP. If keep-breakpoints? is #t (which it is by default), all breakpoints are retained. If keep-breakpoints? is #f, all breakpoints are removed.

4.1.2.11 restart

```
(c-mmp 'restart &key keep-breakpoints?)
```

Reset the MMP and keep breakpoints based on the setting of keep-breakpoints? (see the 'reset method). After resetting the MMP, reload the last object file or the last debugger file that was loaded.

4.1.2.12 select-processor

```
(c-mmp 'select-processor i)
```

Select the specified MPE making it the default MPE and binding it to the symbols *mpe* and &p.

4.1.2.13 write-image-to-file

```
(c-mmp 'write-image-to-file name &optional x-size
 y-size &key base mode)
```

Write an image from display memory to a .PCX file.

The x-size and y-size parameters default to the display height and width. The base defaults to the start of SDRAM and the mode defaults to *display-mode*.

4.1.2.14 write-raw-image-to-file

```
(c-mmp 'write-raw-image-to-file name &optional x-size
 y-size &key base mode)
```

Write an image from display memory to a .pcx file. The x-size and y-size parameters default to the display height and width. The base defaults to the start of SDRAM and the mode defaults to *display-mode*. This method differs from write-image-to-file in that no color space conversion is performed; the Y component of colors is written into the green channel of the output image, Cr into the red, and Cb into the blue.

4.2 C-MPE Class

The C-MPE class is an abstraction of a single Nuon Processing Element (MPE). There is one instance of C-MPE for each MPE on the chip or emulated chip being debugged. The currently selected instance of C-MPE is bound to the variables &p and *mpe*.

4.2.1 Global Variables

- ***mpe*** is always set to the currently selected instance of C-MPE
- **&p** is a synonym for &P

4.2.2 Methods

4.2.2.1 mmp

```
(c-mpe 'mmp)
```

Returns the MMP that contains this MPE.

4.2.2.2 unit-number

```
(c-mpe 'unit-number)
```

Returns the MPE unit number.

4.2.2.3 select

```
(c-mpe 'select)
```

Select this MPE setting the global variables *mpe* and &p.

4.2.2.4 pc

```
(c-mpe 'pc)
```

Return the current value of pcexec for this MPE.

4.2.2.5 fp

```
(c-mpe 'fp)
```

Return the current value of the C frame pointer (r30) for this MPE.

4.2.2.6 fetch-scalar

```
 (c-mpe 'fetch-scalar addr)
```

Returns the scalar at the specified address in MPE memory.

4.2.2.7 fetch-data-scalar

```
(c-mpe 'fetch-data-scalar addr)
```

Returns the scalar at the specified address in MPE memory looking through the data cache.

4.2.2.8 fetch-instruction-scalar

```
(c-mpe 'fetch-instruction-scalar addr)
```

Returns the scalar at the specified address in MPE memory looking through the instruction cache.

4.2.2.9 store-scalar

```
(c-mpe 'store-scalar! addr value)
```

Stores the specified value into the specified address in MPE RAM.

4.2.2.10 store-data-scalar

```
(c-mpe 'store-data-scalar! addr value)
```

Stores the specified value into the specified address in MPE RAM looking through the data cache.

4.2.2.11 store-instruction-scalar

```
(c-mpe 'store-instruction-scalar! addr value)
```

Stores the specified value into the specified address in MPE RAM looking through the instruction cache.

4.2.2.12 read-scalars-from-file

```
(c-mpe 'read-scalars-from-file addr count port)
```

Reads scalars from the specified input port and writes them to the specified address. The data should be in binary form in Nuon (big-endian) byte order.

4.2.2.13 write-scalars-to-file

```
(c-mpe 'write-scalars-to-file addr count port)
```

Reads scalars from the specified address and writes them to the specified output port. The data will be in binary form in Nuon (big-endian) byte order.

4.2.2.14 translate-data-address

```
(c-mpe 'translate-data-address addr)
```

Return the address in the data cache where the specified address is mapped. If the specified address is not in the data cache, return #f.

4.2.2.15 translate-instruction-address

```
(c-mpe 'translate-instruction-address addr)
```

Return the address in the instruction cache where the specified address is mapped. If the specified address is not in the instruction cache, return #f.

4.2.2.16 set-source-path

```
(c-mpe 'set-source-path! path)
```

Sets the path the debugger uses to find source files. The path should be in the form of a list of strings naming directories where files are to be found. The directory separator should be the forward slash (/) even on Windows 95/98 systems.

4.2.2.17 disassemble

```
(c-mpe 'disassemble addr count &optional port)
```

Disassemble instructions starting at the specified address. Instructions are disassembled and printed to the specified port until count bytes have been processed. The default port is *standard-output*.

4.2.2.18 register-address

```
(c-mpe 'register-address name)
```

Return the address of a register.

4.2.2.19 display

```
(c-mpe 'display &optional stream)
```

Display the state of the mpe to the specified stream which defaults to *standard-output*.

4.2.2.20 runtime-eval

```
(c-mpe 'runtime-eval expr)
```

Evaluate the specified expression within an environment where *mpe* and &p are bound to the MPE and the MPE registers are bound to their names (r0, r1, etc.).

4.2.2.21 find-register-by-name

```
(c-mpe 'find-register-by-name name)
```

Returns the register with the specified name. The name should be a string and is case sensitive.

4.2.2.22 find-register-by-address

```
(c-mpe 'find-register-by-address addr)
```

Return the instance of c-register associated with the register with the specified address.

4.2.2.23 register-value

```
(c-mpe 'register-value name)
```

Return the value of the register with the specified name.

4.2.2.24 set-register-value

```
(c-mpe 'set-register-value! name value)
```

Set the value of the register with the specified name.

4.2.3 Execution Control Methods

4.2.3.1 running

```
(c-mpe 'running?)
```

Return #t if the MPE is running and #f otherwise.

4.2.3.2 run

```
(c-mpe 'run)
```

Start the MPE running. Instructions will be executed each time the MPE is clocked.

4.2.3.3 step

```
(c-mpe 'step)
```

Cause the MPE to execute a single instruction.

4.2.3.4 step-over

```
(c-mpe 'step-over)
```

Cause the MPE to skip over a subroutine call.

4.2.3.5 stop

```
(c-mpe 'stop)
```

Stop the MPE from running.

4.2.3.6 wait-for-halt

```
(c-mpe 'wait-for-halt)
```

Wait for the MPE to halt after run or single step.

4.2.4 Breakpoint Methods

Breakpoints are represented by instances of the class **c-breakpoint**. See below for a definition of that class.

4.2.4.1 find-breakpoint

(c-mpe 'find-breakpoint addr)

Returns the instance of c-breakpoint associated with the breakpoint at the specified address. If there is no breakpoint at the specified address, #f is returned.

4.2.4.2 breakpoint

(c-mpe 'breakpoint? addr)

Is there a breakpoint at the specified address? The addr parameter can be either an address or a symbol name passed as a string. Returns true if there is a breakpoint at the specified address and #f if not.

4.2.4.3 set-breakpoint

(c-mpe 'set-breakpoint! addr &key condition count)

Set a breakpoint at the specified address. If the address is a symbol name, the value of the symbol is used as the address. For example:

```
(&p 'set-breakpoint! "loop" :condition
 '(< (r1 'value) 2)))
```

This will break at the label "loop" if the value of r1 is less than 2.

4.2.4.4 clear-breakpoint

(c-mpe 'clear-breakpoint! addr)

Clear the breakpoint at the specified address. If the address is a symbol name, the value of the symbol is used as the address.

4.2.4.5 clear-all-breakpoints

(c-mpe 'clear-all-breakpoints!)

Clear all breakpoints associated with this MPE.

4.2.4.6 map-over-breakpoints

(c-mpe 'map-over-breakpoints fcn)

Applies the specified function to each active breakpoint passing the associated instance of **c-breakpoint** as a parameter. Returns the list of function values. For example:

```
(&p 'map-over-breakpoints (lambda (b) (format #t
"~%Breakpoint at ~X" (b 'address))))
```

This will display the address at which each of the current breakpoints is set.

4.2.4.7 show-breakpoints

(c-mpe 'show-breakpoints)

Show all breakpoints associated with this MPE in the console window.

4.2.4.8 add-before-method

(c-mpe 'add-before-method! addr method)

Add a before method at the specified address. If the address is a symbol, the value of the symbol is used as the address. For example:

```
(&p 'add-before-method! "loop" '(format #t
 "~%At start of loop"))
```

This will display "At start of loop" before the instruction at "loop" is executed.

4.2.4.9 remove-before-method

(c-mpe 'remove-before-method! addr)

Remove the before method at the specified address.

4.2.4.10 add-after-method

(c-mpe 'add-after-method! addr method)

Add an after method at the specified address. If the address is a symbol, the value of the symbol is used as the address. This method works the same as 'add-before-method! except that the method is evalulated after the instruction is executed rather than before.

4.2.4.11 remove-after-method

(c-mpe 'remove-after-method! addr)

Remove the after method at the specified address.

4.2.5 Object File Access Methods

4.2.5.1 set-current-block

```
(c-mpe 'set-current-block! addr)
```

When debugging a C or C++ program, this function sets the current scope to the block containing the specified address. Local symbols are resolved relative to this scope.

4.2.5.2 get-local-symbol-names

```
(c-mpe 'get-local-symbol-names)
```

Get the list of local symbol names in the current scope.

4.2.5.3 find-symbol

```
(c-mpe 'find-symbol name)
```

Find a symbol in the current scope. Returns four values: the symbol value, the overlay identifier, the storage class, and a type specifier. The overlay identifier is #f if the symbol is not in an overlay.

Symbol Classes:

• frame	for variables on the stack
• address	for variables in memory

- register for variables in registers

Type Specifiers:

- void
- char
- short
- int
- long
- float
- double
- unsigned-char
- unsigned-short
- unsigned-int
- unsigned-long
- (struct *tag-name*)
- (union *tag-name*)
- (enum *tag-name*)
- (pointer *type-specifier*)
- (function *type-specifier*)
- (array *size element-size type-specifier*)

4.2.5.4 find-type

```
(c-mpe 'find-type name)
```

Find a type defined in the current scope. Returns a type specifier.

4.2.5.5 get-tag-members

```
(c-mpe 'get-tag-members name)
```

Get the structure or union members associated with the specified name. Returns a list of tag specifiers. Each tag specifier is a list containing the name of the tag, the byte offset from the start of the structure or union (or bit offset for bit fields), a type specifier and a size for bit fields.

4.2.5.6 get-file-reference

```
(c-mpe 'get-file-reference n)
```

Get the specified file reference. File references are number starting at zero and continuing to the number of files minus one. Returns the filename associated with the specified file reference number.

4.2.5.7 get-file-references

```
(c-mpe 'get-file-references)
```

Get a list of all file references. Returns a list of all referenced filenames.

4.2.5.8 find-line-number

```
(c-mpe 'find-line-number addr &optional offset)
```

Find the line number information associated with a specified address. If the **offset** parameter is #t, the line number returned may be associated with an address less than the specified address. It defaults to #f. Returns the file reference number, the line number, the line count and the byte offset from the specified address (or #f if the offset parameter is #f).

4.2.5.9 find-address-from-line-number

```
(c-mpe 'find-address-from-line-number file line)
```

Finds the address associated with the specified file reference number and line number.

4.2.5.10 find-function

```
(c-mpe 'find-function addr)
```

Find the function containing the specified address. Returns the file reference number and function name.

4.2.5.11 start-address

```
(c-mpe 'start-address)
```

Returns the start address specified in the object file.

4.2.5.12 set-start-address

```
(c-mpe 'set-start-address! addr)
```

Set the start address of an MPE.

4.3 C-REGISTER Class

Instances of the C-REGISTER represent registers in the MPE.

4.3.1 Methods

4.3.1.1 name

```
(c-register 'name)
```

Return the name of a register.

4.3.1.2 address

```
(c-register 'address)
```

Return the address of the register.

4.3.1.3 value

```
(c-register 'value)
```

Return the value of the register.

4.3.1.4 set-value

```
(c-register 'set-value! value)
```

Set the value of a register.

4.4 C-BREAKPOINT Class

Instances of the **c-breakpoint** class represent breakpoints set in an MPE.

4.4.1 Methods

4.4.1.1 address

```
(c-breakpoint 'address)
```

Return the address at which the breakpoint is set.

4.4.1.2 symbol

```
(c-breakpoint 'symbol)
```

Return the symbol used to set the breakpoint if there was one. The symbol is returned as a string. Returns #f if no symbol was used to set the breakpoint.

4.4.1.3 breakpoint

```
(c-breakpoint 'breakpoint?)
```

Return #t if there is a user breakpoint at this address.

4.4.1.4 condition

```
(c-breakpoint 'condition?)
```

Return true if there is a condition on this breakpoint. Otherwise, return #f.

4.4.1.5 settings

```
(c-breakpoint 'settings)
```

Return a keyword/value list with the breakpoint settings.

4.4.1.6 change

```
(c-breakpoint 'change! &key breakpoint? condition
count
 before after)
```

Change the settings of the breakpoint.

This page intentionally left blank!

5. Utility Functions

5.1 Fixed Point Math Operations

5.1.1.1 real->32bits

```
(real->32bits value &key fracbits)
```

Convert a real value to a 32 bit fixed point value with the specified number of fracbits. The fracbits parameter defaults to 16.

5.1.1.2 32bits->real

```
(32bits->real value &key fracbits)
```

Convert a 32 bit fixed point value with the specified number of fracbits to a real. The fracbits parameter defaults to 16.

5.1.1.3 64bits->real

```
(64bits->real value-high value-low &key fracbits)
```

Convert a 64 bit fixed point value with the specified number of fracbits to a real. The fracbits parameter defaults to 32.

This page intentionally left blank.