

HAFTA Overlord

P-Code Definition

Copyright ©2002-2003, Astra Network Inc. All Rights Reserved

January 3, 2003

This is documentation for a BETA version of HAFTA. Please be advised that you may encounter problems; please report them so that they may be fixed. The latest information about HAFTA is always available at <http://www.astranetwork.com/hafta/>.

Contents

1	Introduction	3
2	Virtual Machine	3
2.1	The Stack	3
2.2	The Nebuloid Dictionaries	3
2.3	The P-Code Segment Table	4
2.4	The Alarm Stack	4
2.5	The Sigusr Stacks	4
3	P-Code File Format	4
3.1	Version Code	4
3.2	Ola Version Code	4
3.3	Olc Version Code	5
3.4	Stack Size and Policy	5
3.5	Context Count	5
3.6	Module Table	5
3.7	P-Code Segment Table	5
3.8	The P-Code	5
4	Header Codes	6

4.1	olc_ver	6
4.2	proto	6
4.3	stack	6
5	Opcodes	6
5.1	Named Opcodes	7
5.1.1	call	7
5.1.2	alarm	7
5.1.3	signal	7
5.1.4	sigchild	7
5.1.5	get	7
5.1.6	set	8
5.1.7	getpid	8
5.1.8	setpid	8
5.1.9	return	8
5.1.10	inst	8
5.1.11	dinst	8
5.1.12	drop	8
5.1.13	dup	9
5.2	Unnamed Opcodes	9
5.2.1	context	9
5.3	Symboled Opcodes	9
5.3.1	IF / FI	9
5.3.2	FOR / ROF	9
5.3.3	Binary Logical Opcodes	10
5.3.4	Binary Arithmetical Opcodes	10
5.3.5	Unary Opcodes	10
5.4	Reference Constructs	10
5.4.1	mark	11
5.4.2	reference	11
5.4.3	context	11
5.4.4	module	11

5.5	Constant constructs	11
5.5.1	String construct	11
5.5.2	Integer construct	12

1 Introduction

This document defines the format of the system specification file used by the HAFTA overlord runtime (olrt).

The system specification is pcode in either an unassembled or assembled form. Both are defined here.

This is a very low-level specification, most HAFTA users will not need to understand this. It can be useful when debugging but is not necessary.

2 Virtual Machine

The p-code executes in a fairly simple RPN stack-based virtual machine.

Data constructs are stored inline with the opcodes. When a data construct is encountered, it is simply pushed onto the stack. As the opcodes are encountered they pop their arguments from the stack, do whatever calculations they're expected to do, and push their results back on the stack (if appropriate).

Other elements include the Nebuloid Dictionaries, and the P-Code Segment Table.

2.1 The Stack

It's just a stack. Nothing to see here, move along.

2.2 The Nebuloid Dictionaries

The nebuloid dictionaries map nebuloid names to nebuloid instances and their respective modules. They also contain the pid associated with the current dictionary.

There are two kinds of nebuloid dictionaries, the global dictionary (only one), and the local dictionaries (many). The global dictionary is always active and is searched after the local dictionary, of which only one may be active at any given time.

Local nebuloid dictionaries are selected using the context construct (ie: %foo).

Nebuloids are added and removed from the dictionaries using the `inst` and `dinst` opcodes. They are accessed using the `get` and `set` opcodes.

The pid is accessed using the `getpid` and `setpid` opcodes.

2.3 The P-Code Segment Table

The P-Code Segment Table is a fairly standard jump table used for accessing various p-code segments.

2.4 The Alarm Stack

As `alarm` opcodes are encountered, alarm elements are inserted into the alarm stack in the order in which they will be triggered.

When an alarm is triggered, the associated element is removed from the alarm stack and executed.

2.5 The Sigusr Stacks

As `signal` opcodes are encountered, signal elements are appended to the end of either the *sigusr1* or *sigusr2* signal stacks.

When a signal is triggered, the contents of the respective signal stack are popped in order and executed.

While a signal stack is being cleared, new signal elements are appended to a temporary stack which is swapped onto the signal stack only after the signal stack has been cleared and reset.

3 P-Code File Format

The P-Code is stored in big-endian format on disk. It contains a version code, an assembler version code, a compiler version code, a stack size and policy, a context count, a module table, a p-code segment table, and the p-code itself.

3.1 Version Code

32-bit, unsigned, big-endian.

The Version Code defines the p-code version contained in the file. This document defines version 1 of the pcode.

3.2 Ola Version Code

32-bit, unsigned, big-endian.

The Ola Version Code indicates the version of the assembler which was used to create the file.

3.3 Olc Version Code

32-bit, unsigned, big-endian.

The Olc Version Code indicates the version of the compiler which was used to produce the input for the assembler.

3.4 Stack Size and Policy

2 16-bit, unsigned, big-endian.

The Stack Size entry (the first 16-bit field) defines what size stack to allocate upon initialization.

The Stack Policy entry (the second 16-bit field) defines what to do when the stack size is exceeded.

Currently the two policies are

- 0 - grow the stack
- 1 - die horribly

3.5 Context Count

32-bit, unsigned, big-endian.

The Context Count defines how many local nebuloïd contexts are referenced in the pcode.

3.6 Module Table

8-bit, null-terminated strings, empty string terminated.

The Module Table lists the modules referenced by the p-code.

It is used to ensure that all referenced modules are present at module load time. It is also used to enumerate module names.

3.7 P-Code Segment Table

32-bit, unsigned, big-endian. 8-bit, null-terminated strings.

The P-Code Segment Table lists p-code segments as a series of entries consisting of a 32-bit unsigned offset from the start-of-file followed by a string name. It is terminated by a null entry (32-bit zero, and a null string).

3.8 The P-Code

mostly 64-bit, mostly signed, big-endian

The P-Code itself is stored in big-endian format on disk and starts immediately following the P-Code Segment Table. It is terminated by the end-of-file.

4 Header Codes

Header codes are used to declare out of band data in the pcode ASCII format. They can appear anywhere in the pcode, though it is recommended, for clarity's sake, to put them at the start.

Header codes appear as `:type: data :` in the pcode.

Currently there are 3 header codes defined.

4.1 `olc_ver`

`:olc_ver: version :`

The `olc_ver` header code indicates the version of the overlord compiler used to generate the pcode.

4.2 `proto`

`:proto: module , "static args" , "dynamic args" :`

The `proto` header code indicates the static and dynamic arguments of a module. It is not currently required.

The args are represented with an `s` for a string arg, and an `n` for a numeric arg.

4.3 `stack`

`:stack: size , policy :`

The `stack` header directive indicates the size of the initial stack, and the policy if that size is exceeded.

The policy can either be `die` or `grow`.

5 Opcodes

There are 13 named opcodes, 1 unnamed opcodes, 15 symbolized opcodes, 4 reference constructs, a static string construct, and a static integer construct.

Comments appear in the unassembled pcode as a semicolon, followed by the comment, and terminated by an eol character.

5.1 Named Opcodes

Named opcodes appear in the unassembled p-code as keyword names, and appear in the assembled p-code as char sized unsigned integers.

The definitions of the named opcodes are as follows.

5.1.1 call

/ref call #retval

The **call** opcode invokes a p-code segment. Upon completion, the return value of the segment is pushed on the stack.

5.1.2 alarm

#delay /ref alarm

The **alarm** opcode invokes a p-code section after the specified delay.

5.1.3 signal

#sig /ref signal

The **signal** opcode invokes a p-code section upon the overlord receiving the specified signal.

Valid signals are *#1 (SIGUSR1)* and *#2 (SIGUSR2)*.

Note: The handler is cleared once the signal is triggered.

5.1.4 sigchild

/ref sigchild

the **sigchild** opcode invokes a p-code section when the overlord receiving the sigchild signal and the **waitpid()** call returns a pid matching the pid in the current context.

Note: The handler is cleared once the signal is triggered.

5.1.5 get

args .. args (nebuloid) get returnval

The **get** opcode invokes a nebuloid's **get** function with the provided args. Once completed, the return value from the nebuloid is pushed onto the stack.

5.1.6 set

value (nebuloid) set

The **set** opcode invokes a nebuloid's **set** function with the provided value.

5.1.7 getpid

getpid #pid

The **getpid** opcode retrieves the pid associated with the current context and places it on the stack.

5.1.8 setpid

#pid setpid

The **setpid** opcode associates the provided pid with the current context.

5.1.9 return

#retval return

The **return** opcode stops execution of the current p-code segment, and returns to executing the calling segment if the segment was called from pcode, or waits for an alarm/signal to retrigger execution.

The *#retval* argument is pushed back onto the stack after it is trimmed if the segment was called from pcode, otherwise it is discarded.

5.1.10 inst

args .. args #local ℰmodule (nebuloid) inst

The **inst** opcode instantiates an instance of a nebuloid. It binds a module to a nebuloid name in either the global or local context (depending if the *local* arg evaluates as true or false), and passes the provided args to the nebuloid's instantiation routine.

5.1.11 dinst

(nebuloid) dinst

The **dinst** opcode de-instantiates an instance of a nebuloid.

5.1.12 drop

arg drop

The **drop** opcode takes one argument and then ignores it. Used to pop an unneeded item off the stack.

5.1.13 **dup**

arg dup arg arg

The **dup** opcode takes one argument and pushes a second copy to the stack.

5.2 Unnamed Opcodes

There is currently only one unnamed opcode.

5.2.1 **context**

#ctx context

The **context** opcode does not directly appear in the unassembled p-code. It is evaluated from the context reference construct, described below.

5.3 Symboled Opcodes

Symboled opcodes are represented in the unassembled p-code as single non-alphanumeric characters, and appear in the assembled p-code as char sized unsigned integers.

5.3.1 **IF / FI**

*#boolean {
} #boolean*

The **{** (if) opcode takes one argument from the stack. If the argument evaluates as true, the code following the opcode is executed, otherwise all code until the matching **}** (fi) opcode is skipped.

The **}** (fi) opcode terminates an if block, and pushes the logical negation of the originally if'ed value onto the stack.

5.3.2 **FOR / ROF**

*[
#boolean]*

The **[** (for) opcode marks the beginning of the loop.

The **]** (rof) opcode takes one argument from the stack. If the argument evaluates as true, the flow of execution jumps to the matching **[** (for) opcode, otherwise the code following the **]** (rof) opcode is executed.

5.3.3 Binary Logical Opcodes

#arg1 #arg2 oper #result

The binary logical opcodes take two arguments and evaluate them in the form:

result = arg2 oper arg1

These opcodes are as follows:

< less than

> greater than

= equal

& and

— or

5.3.4 Binary Arithmetical Opcodes

#arg1 #arg2 oper #result

The binary arithmetical opcodes take two arguments and evaluate them in the form:

result = arg1 oper arg2

These opcodes are as follows:

+ add

- subtract

* multiply

/ divide

5.3.5 Unary Opcodes

#arg oper #result

The unary opcodes are as follows:

! logical negate

~ arithmetic negate

5.4 Reference Constructs

Reference constructs are higher level functions of the unassembled p-code language that make writing p-code by hand a little bit easier.

There are 4 reference constructs, which are as follows.

5.4.1 mark

p-code: `@string`

compiled: *[nothing]*

The mark construct marks the start of a p-code segment. It does not have an inline representation in the compiled p-code, instead it is represented in a p-code segment table.

5.4.2 reference

p-code: `/string`

compiled: *#index*

The reference construct references the start of the named p-code segment. It is represented by a numeric index into the segment table in the compiled p-code.

5.4.3 context

p-code: `%string`

compiled: *#index context*

The context construct changes the current local nebuloid context. It is represented by a numeric index into the context table followed by the unnamed `context` opcode in the compiled p-code.

5.4.4 module

p-code: `&string`

compiled: *#index*

The module construct references a module by name, for use as an argument in the `inst` opcode.

All module constructs appearing in the p-code are collected into the module table, which is used to determine the index number.

5.5 Constant constructs

Strings and numbers.

5.5.1 String construct

(Some String)

The string construct is represented by an open bracket `(`, followed by the string itself, then terminated with a close bracket `)`. It is copied verbatim into the

compiled p-code, brackets included.

5.5.2 Integer construct

#12345

The integer construct is represented by a pound (octothorpe) character #, followed by a signed 64-bit value. It is represented in the compiled p-code as an integer opcode followed by 64-bits of signed integer.