

HAFA Overlord Language 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 and Basic Concepts	2
1.1	Nebuloids	2
1.2	Procedures	2
1.3	Processes	2
1.4	The System	3
2	Defining the System	3
2.1	Including files	3
2.2	Defining Macros	3
2.3	Setting stack size and policy	4
2.4	Module declarations	4
2.5	SYSTEM	5
2.6	PROCESS	5
2.7	START, RESTART, and DEPENDFAIL	5
2.8	Nebuloids	5
3	Scripting Commands	6
3.1	Nebuloids	6
3.2	Procs	7

3.3	Processes	7
3.4	Expressions	7
3.5	IF	8
3.6	RETURN	8
3.7	Comments	8

1 Introduction and Basic Concepts

The Overlord language is designed to allow the user to define a system—what processes should be running, how to start them, what to do if they fail, etc.—and to tell the Overlord what it needs to do in order to keep this system running smoothly. To this end, it utilizes the concept of processes, procedures (procs), and nebuloids. These concepts are defined in the following sections.

1.1 Nebuloids

Perhaps the most interesting concept of the Overlord language, nebuloids are a common, simple interface to Overlord plugin modules, and provide most of the power of Overlord language scripts. They can be thought of much like variables—in fact, one of the major nebuloid modules *is* the **VAR** module—except with the ability to perform functions when read, such as checking the memory usage of a process. How exactly a nebuloid behaves is based entirely on the corresponding nebuloid module implementation.

Nebuloids can be assigned to and read from, though not every module supports these abilities. Every time a nebuloid is read from, it generally performs whatever function the module was designed to do. Many modules also require that the nebuloid be instantiated, in order to give the module context about what task exactly it is to be performing. For example, when the **LOG** module is instantiated, it is told where it will be logging to (stderr, a file, or the syslog). When the **LOG** nebuloid is read from, it takes the message to be logged and the severity of the log message as arguments.

1.2 Procedures

Procs are simply a collection of Overlord scripting commands which can be called at specific intervals or by other procs, etc. (As we will see later, there are many other structures which can contain Overlord scripting commands; procs can be called by these, as well.) Scripting commands are discussed later on in this article.

1.3 Processes

A process definition in the Overlord language corresponds directly with a process running in a UNIX-like environment. The process definition tells the overlord

how to start a process, what to do when it dies, and contains nebuloids and procs to aid in determining whether or not that process is misbehaving.

Processes also have the concept of dependencies, which are simply other processes which must be running in order for that process to function properly. It is ensured that these dependencies are running when the process is started, and the process is notified if one of the other processes it depends on fails. The user can determine the best course of action in this case, whether it be to simply kill off the process and let the Overlord restart it, to simply ignore it and let the process itself deal with it, or to attempt to notify the process in some way that its dependency is temporarily unavailable.

1.4 The System

A system simply is a collection of processes, procs, and nebuloids (collectively known as *policy*). The procs and nebuloids defined here can be accessed system-wide by scripting commands. (Procs and nebuloids defined in processes can only be accessed within their own process.) The system also has the concept of dependencies; it must be told what processes must be running for the system to function.

2 Defining the System

Now that we have a basic understanding of the language's underlying concepts, it's time to get down into the actual syntax. Everything in the language is case-insensitive; for stylistic purposes, in our examples, actual language keywords and outside module names are in all-caps, while names of procs, nebuloids, etc. are all lowercase.

2.1 Including files

It is possible to include another source file anywhere within your script. Simply use the keyword:

```
INCLUDE "filename"
```

where `filename` is the name of the file you want to include. Files will be included from various directories. In order of preference, the compiler looks in those listed on the command line with the `-I` option, listed in `OLCINCLUDE` environment variable, the current directory, and then `/opt/hafta/include`.

2.2 Defining Macros

Occasionally it may be useful for clarity to define a macro. This is done using the syntax:

```
DEFINE macro replacementtext
```

where **macro** is the name given to the macro, and **replacementtext** is the text which replaces it when used in code. For example:

```
DEFINE verbosity 5
...
log(verbosity, "Hello!");
```

2.3 Setting stack size and policy

The Overlord runtime uses a stack to execute the compiled scripts. It is conceivable that a script could be written which overruns this stack. Thus the Overlord compiler provides a method of specifying the stack size:

```
STACKSIZE size
```

where **size** is the size of the stack in nodes. This must be placed before the beginning of the **SYSTEM** definition. The default is 256. It is also possible to specify the Overlord's behaviour when the stack is overrun. By default, it will assume that it's running on a constrained system, and die rather than taking up additional resources. However, you can also tell the Overlord to grow the stack when it runs out, using the **STACKPOLICY** directive, like so:

```
STACKPOLICY [DIE | GROW]
```

where **DIE** is used if you want the Overlord to die instead of growing the stack, and **GROW** is used if you want the Overlord to grow its stack as necessary.

2.4 Module declarations

If it is required that some nebuloids be implicitly instantiated, it is required that the modules to be used be declared before the system declaration. The syntax for this is relatively straightforward, it's simply:

```
MODULE name (static argument list) (dynamic argument list)
```

where **name** is, obviously, the name of the module you wish to use in your script.

The **static** and **dynamic** argument lists require a tad more explanation; they are lists of the types of arguments taken during instantiation and actual use. Thus, the declaration for the **LOG** module would look like:

```
MODULE LOG (NUM, STRING) (NUM, STRING)
```

Thankfully, this should only really matter to you if you write your own modules, as we provide a headers for all standard nebuloid types.

2.5 SYSTEM

The bulk of the language script is contained in the **SYSTEM** declaration. This is quite simple; it works as follows:

```
SYSTEM: dependencies
{
    policy
}
```

where **dependencies** is a list of process names within the system which must be running for the system to work, and **policy** is the various procedures, procs, and nebuloids contained within the system.

2.6 PROCESS

Within a **SYSTEM** definition, you can declare processes, using the **PROCESS** keyword. This looks like so:

```
PROCESS name interval: [dependencies]
{
    policy
}
```

name refers to the name used to refer to this process in script code. **interval** is the number of milliseconds between checks to see whether or not this process is still alive and running. **dependencies** is an optional list of other processes which this process requires to be running before it can function. **policy** includes not only proc and nebuloid definitions, but **START**, **RESTART** and **DEPENDFAIL** definitions as well. These are basically special-case procs which are only called by the Overlord; they tell the Overlord how to start the process, what to do when the process dies to restart it, and how to react when one of its dependencies fails, respectively.

2.7 START, RESTART, and DEPENDFAIL

The text for this section has not been completed. Please see the “Getting Started” tutorial for an explanation of these concepts.

2.8 Nebuloids

Many Overlord modules require that you instantiate a nebuloid, usually with some arguments to define exactly what it is it’s supposed to be doing. Nebuloids can be declared either within a system, or within a process. The declaration for nebuloids is as follows:

```
modulename instancename ([argument1 [, argument2 [, ...]]) [interval
{
```

```

    script
  }]
```

modulename is the name of the type of nebuloïd you are instantiating. **instancename** is the name you will use in Overlord script to refer to that nebuloïd. The **arguments** can be either a number or a string literal.

If a nebuloïd is used specifically for checking some condition of the system, it may be desirable to add the script to actually check this condition right in the nebuloïd definition. In this case, you would specify **interval** as the number of milliseconds between each check, and then write the appropriate **script**.

3 Scripting Commands

Often, you will need to specify to the overlord how to check a particular condition, or what to do when something fails. This is done using Overlord script.

All Overlord script statements end with semicolons.

3.1 Nebuloïds

Nebuloïds can be accessed from Overlord script in three ways: as part of an expression, as a statement on its own, or as having a value assigned to it.

To assign a value to a nebuloïd, you simply use the assignment operator:

```
nebuloïd = expression;
```

Where **nebuloïd** is the name of the nebuloïd you are assigning to, and **expression** is a mathematical expression (explained in Section 3.4).

To read a value from a nebuloïd, either to use its value in an expression, or to perform a function (depending on the function of the module), you would use the following form:

```
nebuloïd [(argument1[, argument2[, ...]])];
```

Where **nebuloïd** is, rather obviously, the name of the nebuloïd you wish to access, and the **arguments** are the various arguments passed to that nebuloïd when it is accessed. For more information about what arguments should be passed to various nebuloïds, please refer to the *Nebuloïd Type Reference*.

It should also be noted that if the module does not require arguments to be accessed, the brackets around the argument list are unnecessary. This is for convenience, so that the code

```

...
VAR tmp (0)
PROC inctmp
{
```

```

    tmp = tmp + 1;
}
...

```

works as expected.

In addition, if no arguments are required for their instantiation, and no context is required to be kept between accesses, some nebuloids can be implicitly instantiated. The KILL module is a good example of this. To implicitly instantiate a nebuloid, declare the module at the beginning of the script, then simply use the module name instead of an instantiated nebuloid's name when accessing the nebuloid. For example:

```
KILL ("SIGTERM", process3);
```

3.2 Procs

To call a proc from script code, simply use its name. Since procs must return true or false, these can be used in expressions. Note that you may only call procs which are local to your current process, or global to the entire system.

3.3 Processes

In the **START** condition of a process, there needs to be a way to specify to the Overlord what the pid of the newly-started process is. The Overlord language solves this by allowing process names to behave as variables which contain the pid of that process. This means you could have a system which looked something like this:

```

SYSTEM: process1
{
    PROCESS process1 100: process2
    {
        START { process1 = EXEC ("/bin/process1"); }
        DEPENDFAIL
        { // If process2 fails, kill off process1 and restart it.
            KILL ("SIGTERM", process1);
        }
        ...
    }
    ...
}

```

Magic!

3.4 Expressions

Mathematical expressions work like in C, except there's no "!" operator. If we magically decide we need a "!" operator, it'd be pretty trivial to implement a

“!” operator. Also, an assignment statement doesn’t count as an expression, so no `var1 = var2 = 5;` or funny business like that.

3.5 IF

The IF statement is really simple, it works as follows:

```
IF (expression)
    code
[ELSE code]
ENDIF;
```

expression, of course, is a mathematical expression, zero being false and nonzero being true. The blocks of **code** can be as long as you feel like making them. And due to the clever use of **ENDIF**, no dangling **ELSE** problems! Pretty straightforward.

Note that the IF statement, while containing blocks of other statements, *is* still considered a statement, so **ENDIF** is immediately followed by a semicolon.

3.6 RETURN

Procs n’ such must return true or false. This is done through the use of the following commands:

RETURN TRUE; Returns a “true” value.

RETURN FALSE; Returns a “false” value.

If no return statement is given for the proc, the proc by default returns true.

3.7 Comments

The Overlord languages uses C++-style comments; that is, everything until the end of line after “//” is ignored by the overlord compiler.