א Programming Language

Debugging with Aleph

Volume 3 **Revision 0.9.0**

CONTENTS

Preface

This manual is part of the *Aleph Programming Language Series*, a multi volume set that describes the programming environment of the **Aleph** system. The entire set contains 4 volumes :

Volume 0 - Aleph Installation Guide is the distribution installation manual.

Volume 1 - Aleph Programmer Guide is the first volume of this set. It is both an introduction and an advanced guide for the the developer.

Volume 2 - Aleph Library Reference is the second volume of this set. It is a complete description of the Aleph standard library.

Volume 3 - Aleph Cross Debugger is the third volume of this set. It is a reference manual to develop and debug Aleph programs.

Volume 4 - Aleph C++ API is the fourth volume of this set. It is a reference manual of the C++ Application Programming Interface (API).

The Aleph programming language

Aleph is a multi-threaded functional programming language with dynamic symbol bindings that support the object oriented paradigm. **Aleph** features a state of the art runtime engine that supports both 32 and 64 bits platforms. **Aleph** comes with a rich set of libraries that are designed to be platform independent. **Aleph** is a free software. A flexible license has been designed for both individuals and corporations. Everybody is encouraged to use, distribute and/or modify the aleph engine for any purpose.

Features

The **Aleph** engine is written in C++ and provides runtime compatibility with it. Such compatibility includes the ability to instantiate C++ classes, use virtual methods and raise or catch exceptions. A comprehensive API has been designed to ease the integration of foreign libraries.

- **Builtin objects**
  More than 50 reserved keywords and predicates. Various containers like list, vector, hash table, bitset, and graphs.
- **Functional programming**
  Support for *lambda expression* with explicit closure. Symbol scope limitation with *gamma expression*. Form like notation with an easy block declaration.

- **Object oriented**
  Single inheritance object mechanism with dynamic symbol resolution. Native class derivation and method override. Static class data member and methods.

- **Multi-threaded engine**
  True multi-threaded engine with automatic object protection mechanism against concurrent access. Read and write locking system and thread activation via condition objects.

- **Original regular expression**
  Builtin regular expression engine with group matching, exact or partial match and substitution.

Aleph is a core language and libraries. The libraries are a specifi c set of classes and functions which are structured per application domains. **Aleph** is delivered with a set of standard libraries.

- **aleph-sys**
  The `aleph-sys` library is the system calls library. Standard classes and functions are provided to interact with the running machine.

- **aleph-sio**
  The `aleph-sio` library is the standard input/output All input/output operations are performed with this library.

- **aleph-net**
  The `aleph-net` library is the networking library. The library is based on the standard *Internet Protocol* and provides various classes to manipulates IP address, client or server sockets.

- **aleph-www**
  The `aleph-www` library is the World Wide Web library. The library provides various classes that ease the development of web applications or CGI scripts.

- **aleph-txt**
  The `aleph-txt` library is the text processing library. The library provides various functions and classes that ease text manipulation. Sorting data, computing message digest and formatting table is among others, features available in this library.

- **aleph-odb**
  The `aleph-odb` library is the object database library. The library provides several objects that can be used to design a database. A client is also provided to directly access the database contents.

Aleph provides *extensions*. An extension is a library or an application which is not installed by default. The user selects during the installation process which extension is needed. For example, the static version of the interpreter is an extension.

Aleph engine

Aleph is an interpreted language. When used interactively, commands are entered on the command line and executed when a complete and valid syntactic object has been constructed. Alternatively, the interpreter can execute a source fi le. **Aleph** does not have a garbage collector. **Aleph** operates with a lazy, scope based, object destruction mechanism. Each time an object is no longer visible, it is destroyed automatically. At this time, the **Aleph** interpreter is unable to reclaim memory with circular structures. This is a well known problem when using a reference count mechanism. In the future, the **Aleph** engine will provide some mechanisms to resolve this problem.

Flexible Distribution

Aleph is a free software. A flexible license model encourages individuals or corporations to use, copy, modify and/or distribute this software. **Aleph** is designed by software professionals. Quality is one the driving force of the development effort. This is reflected in this distribution by the extensive documentation. A large test suite is used to assess the quality of the distribution. Right now, the engine has been successfully tested on most Linux platforms, Free BSD and Solaris.

License

Aleph is a free software. It can be used, modified and distributed by anybody for personal or commercial use. The only restriction is altering the copyright notice associated with the material. Individual or corporation are permitted to use, include or modify the **Aleph** engine. All material developed with the **Aleph** language belongs to their respective copyright holder.
This program is a free software. it can be redistributed and/or modified, providing that this copyright notice is kept intact. This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. In no event shall the copyright holder be liable for any direct, indirect, incidental or special damages arising in any way out of the use of this software.

CHAPTER 1
Introduction

This chapter is short introduction to the **Aleph cross debugger** or **axd**. The **Aleph** debugger is a special interpreter that is designed to help the developer to trace an **Aleph** program. The debugger is designed to operate in a stand-alone mode or with `Emacs`. If you plan to use the debugger with Emacs, you will have to install a `gud-mode` for `Aleph`.

1.1 A sample axd session

The **Aleph Cross Debugger** or **axd** is a special interpreter that gives the developer the opportunity to trace an **Aleph** program and examine the object contents during the execution. Operations normally available in a debugger are available with **axd**. Such operations include breakpoints, stepping, stack tracing, and many others. Because **axd** is built on top of the **Aleph** interpreter, all standard operations are supported by the debugger.

1.1.1 Starting the debugger

The debugger is started with the command `axd`. Within Emacs, the command `Meta-x axd` will do the same. When the debugger is started, an `axd` prompt is displayed. At this stage, there is no difference with the standard **Aleph** interpreter, except that a new *nameset* called `axd` is defi ned with all debugger commands. The `axd:quit` or `axd:quit` command will terminate the `axd` session.

```
zsh >axd
(axd)axd:quit
zsh >
```

1.1.2 Debugger commands

All debugger commands are located in the *axd nameset*. For example, the command to set a breakpoint is `axd:break`. Since typing such command can be annoying, it is possible to rebind them at your convenience. For example, the form `const b axd:break` will defi ne the symbol `b` as the breakpoint command, but care should be taken with this approach if your program uses the same symbol.

1.1.3 Debugging an example

The first example that demonstrates the use of **axd** is located in the directory `exp/als`, that is part of this distribution. The platform information example `0501.als` will be used for illustration. A simple **Aleph** session and the original source code is given below.

```
zsh >aleph 0501.als
amaury> aleph 0501.als
major version number   : 0
minor version number   : 9
patch version number   : 0
interpreter version    : 0-9-0
program name           : aleph
operating system name  : linux
operating system type  : unix
aleph official url     : http://www.aleph-lang.org

zsh >less 0501.als
# many comments before
println "major version number   : " interp:major-version
println "minor version number   : " interp:minor-version
println "patch version number   : " interp:patch-version
println "interpreter version    : " interp:version
println "program name           : " interp:program-name
println "operating system name  : " interp:os-name
println "operating system type  : " interp:os-type
println "aleph official url     : " interp:aleph-url
```

The debugger is started with the file to debug. The **info** command can be used to print some information.

```
zsh > axd 0501.als
(axd) axd:info
debugger version    : 0-9-0
os name             : linux
os type             : unix
initial file        : 0501.als
form file name      : 0501.als
form line number    : 17
verbose mode        : true
max line display    : 10
defined breakpoints : 0
(axd)
```

Along with the version, initial file name and other information, is the *form file name* and *form line number* that indicates where the debugger is position. Another way to get this information is with the **list** command that display the file at its current break position.

```
(axd) axd:list
17    println "major version number   : " interp:major-version
18    println "minor version number   : " interp:minor-version
19    println "patch version number   : " interp:patch-version
20    println "interpreter version    : " interp:version
21    println "program name           : " interp:program-name
22    println "operating system name  : " interp:os-name
23    println "operating system type  : " interp:os-type
24    println "aleph official url     : " interp:aleph-url
```

```
25
26
(axd)
```

With this in place it is possible to run the program. The **run** command will do the job, but will not give you the opportunity to do something since there is no breakpoint installed. So, installing a breakpoint is simply achieved by giving the file name and line number. To make life easier, the **break** command takes also 0 or argument. Without argument, a breakpoint is set at the current position. With one integer argument, a breakpoint is set at the specified line in the current file. If the verbose mode is active (which is the default), a message is printed to indicate the breakpoint index.

```
(axd) axd:break 19
setting breakpoint 0 in file 0501.als at line 19
(axd)axd:run
major version number   : 0
minor version number   : 9
breakpoint 0 in file 0501.als at line 19
(axd)
```

The **run** command starts the program and immediately stops at the breakpoint. Note that the debugger prints a message to indicate the cause of such break. After this, stepping is achieved with the **next** command. Resuming the execution is done with the **continue** command. The **exit** or **quit** command terminates the session.

```
(axd)axd:next
patch version number   : 0
(axd)axd:next
interpreter version    : 0-9-0
(axd)axd:continue
program name           : axd
operating system name  : linux
operating system type  : unix
aleph official url     : http://www.aleph-lang.org
(axd)axd:quit
zsh >
```

CHAPTER 2
Using the Debugger

This chapter describes in detail the usage of the **Aleph Cross Debugger** or **Axd**. The debugger is a special application that is built on top of the **Aleph** interpreter. For this reason, the debugger provides the full execution environment with special commands bound into a dedicated nameset.

2.1 Invocation and termination

Axd is started by typing the command `axd`. Once started, the debugger reads the commands from the terminal. Since the debugger is built on top of the **Aleph** interpreter, any command is in fact a special form that is executed by the interpreter. The natural way to invoke the debugger is to pass the primary file to debug with eventually some arguments.

```
zsh> axd PROGRAM [arguments]
```

When the debugger is started, a prompt '(axd)' indicates that the session is running. The debugger session is terminated with the commands `exit` or `quit`.

```
zsh> axd PROGRAM
(axd) axd:quit
zsh>
```

2.2 Options

The available options can be seen with the '-h' option and the current version with the '-v' option. This mode of operations is similar to the one found with the **Aleph** interpreter.

```
zsh> axd -h
usage: axd [options] [file] [arguments]
           [-h]             print this help message
           [-v]             print version information
           [-i] path        add a path to the resolver
           [-f  assert]     enable assertion checking
           [-f  emacs]      enable emacs mode
zsh>
```

2.3 Running the program

When a program is run with **Axd**, a primary file must be used to indicate where to start the program. The file name can be given either as an **Axd** command argument or with the `axd:load` command. The first available form in the primary file is used as the program starting point.

2.3.1 Program loading

The `axd:load` command loads the primary file and mark the first available form as the starting form for the program execution. The command takes a file name as its first argument. The **Aleph** resolver rule apply for the file name resolution.

- If the string name has the '.als' extension, the string is considered to be the file name.
- If the string name has the '.axc' extension or no extension, the string is used to search a file that has a '.als' extension or that belongs to a librarian.

Note that these operations are also dependent on the '-i' option that adds a path or a librarian to the search-path.

2.3.2 Starting the program

The `axd:run` command starts the program at the first available form in the primary file. The program is executed until a breakpoint or any other halting condition is reached. Generally, when the program execution is suspended, an entry into the debugger is done and the prompt is shown at the command line.

```
(axd)axd:run
```

The `axd:run` is the primary command to execute before the program can be debugged. Eventually, a file name can be used as the primary file to execute.

```
(axd)axd:run "test.als"
```

2.3.3 Setting program arguments

Since the debugger is built on top of the **Aleph** interpreter, it is possible to set directly the argument vector. The argument vector is bound to the interpreter with the qualified name `interp:argv`. The standard vector can be used to manipulate the argument vector.

```
(axd)interp:argv:reset
(axd)interp:argv:append "hello"
```

In this example, the interpreter argument vector is reset and then a single argument string is added to the vector. If one wants to see the interpreter argument vector, a simple procedure can be used as shown below.

```
const argc (interp:argv:length)
loop (trans i 0) (< i argc) (i:++) {
  trans    arg (interp:argv:get i)
  println "argv[" i "] = " arg
}
```

2.4 Breakpoints operations

Breakpoints are set with the `axd:break` command. If a breakpoint is reached during the program execution, the program is suspended and the debugger session is resumed with a command prompt. At the command prompt, the full interpreter is available. It permits to examine symbols.

2.4.1 Breakpoint command

The `axd:break` command sets a breakpoint in a file at a specified line number. If the file is not specified, the primary file is used instead. If the line number is not specified, the first available form in the current file is used.

```
(axd) axd:break "demo.als" 12
Setting breakpoint 0 in file demo.als at line 12
```

In this example, a breakpoint is set in the file "demo.als" at the line number 12. The file name does not have to be the primary file. If another file name is specified, the file is loaded, instrumented and the breakpoint is set.

2.4.2 Breakpoint viewing

The `axd:break-info` command reports some information about the current breakpoint setting.

```
(axd) axd:break "demo.als" 12
(axd) axd:break "test.als" 18
(axd) axd:break-info
Breakpoint 0 in file demo.als at line 12
Breakpoint 1 in file test.als at line 18
```

2.4.3 Breakpoint resume

The `axd:continue` command resumes the program execution after a breakpoint. The program execution continues until another breaking condition is reached or the program terminates.

```
(axd) axd:run
Breakpoint 0 in file demo.als at line 12
(axd) axd:continue
```

In this example, the program is run and stopped at breakpoint 0. The `axd:continue` command resumes the program execution.

APPENDIX A
Debugger commands

This appendix contains the **Aleph cross debugger** command reference. The **Aleph cross debugger** is started with the `axd` command. All commands are bound to the **axd** nameset.

Table 1 Debugger commands

Command	Description
run	run the debugger session
load	load the initial fi le
next	execute next form
info	report debugging information
exit	terminate the debugger session
quit	terminate the debugger session
list	display form listing at current line
break	set a breakpoint
continue	continue execution after a breakpoint
break-info	report breakpoint information

break [axd]

Description

The `break` command sets a breakpoint. Without argument a breakpoint is set in the current file at the current line. With a line number, the breakpoint is set in the current file. With two arguments, the first one is used as the file name and the second one is used as the line number.

Example

```
(axd) axd:break "demo.als"  12
(axd) axd:break 25
```

The first example sets a breakpoint in the file `demo.als` at line 12. The second example sets a breakpoint in the current file at line 25. Without argument, the command sets the breakpoint at the current line. The current line can be seen with the `info` command.

break-info [axd]

Description

The `break-info` command reports some information about the current breakpoints.

Example

```
(axd) axd:break "demo.als" 12
(axd) axd:break "test.als" 18
(axd) axd:break-info
Breakpoint 0 in file demo.als at line 12
Breakpoint 1 in file test.als at line 18
```

In this example, two breakpoints are set. One in file 'demo.als' at line 12 and one in file 'test.als' at line 18. The `break-info` command reports the current breakpoint settings.

continue [axd]

Description

The `continue` command resumes the program execution after a breakpoint. The program execution continues until a breakpoint or another terminating condition is reached.

```
(axd) axd:run
Breakpoint 0 in file demo.als at line 12
(axd) axd:continue
```

In this example, the program is run and stopped at breakpoint 0. The `axd:continue` command resumes the program execution.

exit [axd]

Description

The `exit` command terminates a debugger session. This command is similar to the `quit` command.

info [axd]

Description

The `info` command reports some debugger information. Such information includes the debugger version, the operating system, the primary input file, the primary input file source and more.

Example

```
(axd) axd:info
debugger version    : 0-9-0
os name             : linux
os type             : unix
initial file        : 0501
form file name      : 0501.als
form line number    : 17
verbose mode        : true
max line display    : 10
defined breakpoints : 0
```

list [axd]

Description

The `list` command display the form listing starting at the current session line number. The current form line number can also be seen with the `info` command. The number of line is a debugger parameter. The first line to display can also be set as the first parameter. A file name can also be set.

Example

```
(axd) axd:list
(axd) axd:list 20
(axd) axd:list "file.als" 20
```

The first example shows the listing at the current debugger line. The second example starts the listing at line 20. The third example starts at line 20 with file "file.als".

load [axd]

Description

The `load` command sets the *initial or default* file to be used with the `run` command.

Example

```
(axd) axd:load "demo.als"
```

In this example, the file 'demo.als' is set as the primary file. Using the `info` command will report at which line, the first available form has been found.

next [axd]

Description

The `next` command executes the next line in the source file. The `next` command does not take argument.

Example

```
(axd) axd:next
```

quit [axd]

Description

The `quit` command terminates a debugger session. This command is similar to the `exit` command.

run [axd]

Description

The `run` command executes the default file in the slave interpreter. Without argument, the *initial or default* file is executed. The `load` command can be used to set the *initial or default* file. With one argument, the file name argument is used as the *initial or default* file.

Example

```
(axd) axd:run
(axd) axd:run "demo.als"
```

The first example runs the initial file. The second example sets the initial file as `demo.als` and run it.

INDEX

Colophon

This manual was written for the LaTeXdocumentation preparation system. A custom document class was designed by the author. The document style has been simplified as to produce a high quality technical manual. Title, chapter and section names have been produced with an Helvetica font. The document has been produced with a 10 points Times font. Both fonts are assumed to be in the public domain. The documentation is available in both A4 and letter format.