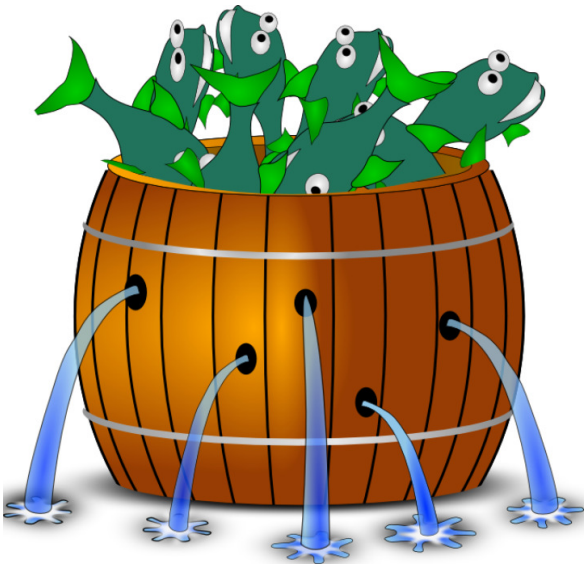
Barrelfish Project
ETH Zurich



Hake

Barrelfish Technical Note 003

Timothy Roscoe

11.04.2010

Systems Group, ETH Zurich
c/o Department of Computer Science
ETH Zurich CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland

`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
1.0	3.06.2010	TR	Initial version
1.1	11.04.2010	TR	Support for out-of-tree builds

Contents

Chapter 1

Introduction

Hake is how we build Barrelfish.

Hake isn't designed to operate outside Barrelfish, so this document will assume you're trying to build Barrelfish.

1.1 Quick start

Suppose you have a fresh Barrelfish source tree in:

`/home/barrelfish/src`

To build a tree, create a directory for it, `cd` to that directory, and run the Hake bootstrap script, and then Make:

```
$ cd /home/barrelfish/src
$ mkdir ../build
$ cd ../build
$ ../src/hake/hake.sh -s ../src
...
$ make -j 32
...
```

Edit the file `symbolic_targets.mk` in your build directory to add extra make targets.

Edit the file `hake/Config.hs` in your build directory and rebuild Hake to reconfigure your build tree.

That's about it.

1.2 How to think about hake

Hake is essentially a Haskell embedded domain-specific language, except that it is also evaluated dyanically (using the `System.Eval.Haskell` package) and written by scattering code around the source tree.

Hake consists of the main hake program (which itself contains considerable information on how to build code), together with a set of Hakefiles spread throughout the source tree.

Each Hakefile should be thought of as containing a Haskell expression which evaluates to a set of rules for Make. The expression will be evaluated in an environment which includes the path to the directory where the Hakefile is located, plus a complete list of all files in the source tree.

1.3 When hake runs

When you run Hake in a Barrelfish source tree, it does the following things:

1. Hake builds a list of (almost) every file and directory in the source tree. The list of files Hake ignores is currently hardcoded into Hake, but basically it skips editor temporary files, version control directories, and products of a previous build process.
2. From this, Hake extracts a list of all Hakefiles in the tree.
3. Hake reads every Hakefile. Each Hakefile contains a single Haskell expression which itself evaluates to a set of Make rules.
4. Hake constructs a single very large Haskell expression out of all these Hakefiles. Each Hakefile is evaluated in an environment which includes the pathname of the Hakefile itself (to resolve relative names), and the entire list is evaluated in an environment which includes the list of files in the whole tree (to allow wildcards).
5. This large expression is then evaluated. The result is a single tree of Hake rule representations (see Chapter 3.2).
6. The rule tree is traversed to derive a list of every directory in the build tree.
7. Finally, a single Makefile is generated which contains rules to build every target in the build tree, for every architecture (including the host-based build tools themsevles), and also create every directory in the build tree.

This single Makefile is large, but is also quite simple: it contains no use of Make variables or generic Make rules, instead it simply includes explicit rules to build every file required for Barrelfish.

1.4 Motivation and Design Principles

Hake should be a full programming language. The lesson from countless built systems is that if one starts without a full programming language built in, one ends up implementing a bad one (CMake being only one example). It's much easier to bite the bullet and admit that we need a complete language, and plenty are available for this.

Hake should be a functional language. `make` is a canonical example of a successful declarative language: Hake should not try and replicate what make does well.

Hake should generate one Makefile. One Makefile is easier to debug: all the information is available in the same file. There is no need to hunt through 5 levels of include files. The only thing Hake-generated Makefiles include are generated C dependency lists, and a single, top-level file giving symbolic targets. The Makefile generated by Hake also makes minimal, and highly stylized, use of make variables: wherever, any variable substitution is done in Haskell before the Makefile is generated.

Hake is for building Barrelfish. We make no claims as to Hake's suitability for any project other than Barrelfish, and indeed the current implementation is pretty tied to the Barrelfish tree. This has helped to keep the system focussed and tractable. One non-goal of Hake, for example, is to support portability across host machines (as CMake tries to do).

Chapter 2

Simple Hakefiles

Hake can in principle build anything, but there are two simple use cases for Hake: building Barrelfish applications (user-space binaries), and building Barrelfish libraries. Here's how, at time of writing, the Barrelfish PCI driver is specified. This is the entire Hakefile:

```
[ build application {
       target = "pci",
       cFiles = [ "pcimain.c", "pci.c", "pci_service.c",
                  "ioapic.c", "acpi.c", "ht_config.c",
                  "acpica_osglue.c", "interrupts.c",
                  "pci_confspace.c", "pcie_confspace.c",
                  "video.c", "buttons.c", "acpi_ec.c" ],
        flounderBindings = [ "pci" ],
        flounderDefs = [ "monitor" ],
        mackerelDevices = [ "pci_hdr0", "pci_hdr1",
                            "lpc_ioapic", "ht_config",
                            "lpc_bridge", "acpi_ec" ],
        addIncludes = [ "acpica/include" ],
        addCFlags = [ "-Wno-redundant-decls" ],
        addLibraries = [ "mm", "acpi", "skb", "pci" ],
        architectures = [ "x86_64", "x86_32" ]
       }
]
```

The outermost square brackets are a Haskell list expression - each Hakefile should be such a list (the exact type will be explained later). This list has a single element, the instruction to built an application (you can have more of these, separated by commas).

The `build application` specifies a number of arguments, all of which are optional. These are actually Haskell record field specifiers, and `application` returns a complete default set. `build` then generates the Make rules to build the application.

The complete list of possible arguments for applications (or libraries) can be found by looking at `Args.hs`. The ones used here are:

target : the name of the binary to build.

cFiles : list of names of C source files. You need to include "`.c`".

flounderBindings : Flounder interfaces for which to compile the stub files.

flounderDefs : Flounder interfaces to use from a library

mackerelDevices : list of Mackerel device specs this application uses or depends on.

addIncludes : additional include paths for header files.

addLibraries : additional libraries to link against.

Note that filenames are relative to the current source directory. Those with a leading '/' are interpreted relative to the root of the tree (not the root file system).

Libraries are similar. Here's the Hakefile for the X86 emulator library:

```
[ build library {
         target = "x86emu",
         cFiles = [ "debug.c", "decode.c", "fpu.c", "ops2.c",
                    "ops.c", "prim_ops.c", "sys.c"],
         addCFlags = ["-Wno-shadow" ]
     }
]
```

This should be all you need to know to write simple Hakefiles for the Barrelfish, and indeed to understand most of the Hakefiles in the Barrelfish tree.

Doing (or understanding) more fancy things in the Hakefile requires more knowledge of how Hake internally generates and represents Make rules, described later.

Chapter 3

Hake from the bottom up

The core of Hake consists of the code to walk the source tree, a minimal set of data types used to represent Make rules, and codes to render these data types into a Makefile.

3.1 The Hake name space for files

Unlike most build systems, Hake uses a 3-dimensional name space for files.

The first component is called the "tree". Whenever Hake runs, it deals with three "trees":

1. The "source tree" (written as `SrcTree`) is the fle system directory tree containing the source code for the programs and libraries currently being built. When building the core OS, this is the main OS source tree. When building an external application or library, this is the directory tree containing the application or library's source code.
2. The "build tree" (written as `BuildTree`) is where the intermediate and final results of the compilation end up. This is typically the current working directory when Hake was run.
3. The "install tree" (written as `InstallTree`) is the directory tree containing a core Barrelfish OS build tree. When building the OS, the install tree and the build tree are the same, but when building an external application or library, the install tree is a pre-built Barrelfish tree and the build tree is where the new application or library is built.

The second component is called the "architecture" (for want of a better name), and corresponds to building the same code for different target architectures (`x86_64`, `arm`, etc.) Architectures themselves form a flat namespace.

Some "architectures" are special when building the core Barrelfish OS:

src refers to files which are always present in the source tree. Hake should not be used to build anything in the `src` architecture, and anything in any other architecture must be generated at build time.

hake is used by Hake as part of the bootstrapping process.

root refers to files relative to the top of the build tree, and should be used with caution.

tools is used to build other build process tools (Mackerel, Fugu, Flounder, etc.)

docs is used to build documentation (Technical Notes), including this document.

The final component is called the "path", and corresponds roughly to the pathname of the file from the root of the designated tree. In the source for hake itself,"path" usually refers to this path, and file "location" or just "loc" refers to the 3-dimensional file reference.

Here are some examples of hake file locations:

Tree	Architecture	Path	Description
SrcTree	src	/tools/hake/Main.hs	Part of the source code for Hake itself
InstallTree	tools	/tools/flounder/flounder	The (built) binary for the flounder compiler
InstallTree	x86_64	/lib/libbarrelfish.a	The Barrelfish library
InstallTree	src	/include/stdio.h	C header file
BuildTree	x86_64	/include/asmoffsets.h	Generated C header file
SrcTree	src	/devices/xapic.dev	Mackerel source file
BuildTree	x86_64	/include/dev/xapic_dev.h	Generated header file from Mackerel

When referring to files in Hake, files whose paths are "relative" (i.e. do not start with a leading "/") are considered relative to the path of their Hakefile, and are converted into "absolute" paths from the top of their tree when they appear. This is more intuitive than it sounds. For example, a file referred to as `(SrcTree,``src'',''e1000.c'')` in `drivers/e1000/Hakefile` will appear in the resulting Makefile as drivers/e1000/e1000.c.

The Hake namespace is mapped onto the file system as follows: all files with architecture `src` are relative to the top of the source tree, whereas a file in a different architecture `foo` is relative to directory `foo/` in the build or install tree. Thus, `(BuildTree,``x86_64'',''e1000.o'')` in `drivers/e1000/Hakefile` will appear in the resulting Makefile as ./x86_64/drivers/e1000/e1000.o.

Hake will generate all Makefile rules necessary to create any directories in the build tree that it needs - it's perfectly possible (and sometimes useful) with Hake to type "`rm -rf ./*; make`" and have everything work.

3.2 Representing rules

Each Hakefile is an expression that must evaluate to a list of `HRules`. The declaration of `HRule` is:

```
data HRule = Rule [ RuleToken ]
           | Include RuleToken
           | Error String
           | Rules [ HRule ]
             deriving (Show,Typeable)
```

The `Include` constructor creates an "include" directive in a Makefile. In theory, there should be no need for developers to use this; it is only used currently to include automatically-generated dependency files for C and assembly source.

The `Rules` constructor allows a tree of rules to be constructed. This is purely a convenience: any time that one can return a single rule, one can also return a list of rules. This makes it easier to write functions which return rules, which is the basis of Hake.

The `Error` constructor is used to signal errors, but in practice is rarely used.

An actual basic Makefile rule is constructed by `Rule` as a list of `RuleTokens`. The declaration of `RuleToken` is:

```
data TreeRef = SrcTree | BuildTree | InstallTree
             deriving (Show,Eq)

data RuleToken = In     TreeRef String String -- Input to the computation
               | Dep    TreeRef String String -- Extra (implicit) dependency
               | NoDep  TreeRef String String -- File that's not a dependency
               | PreDep TreeRef String String -- One-time dependency
               | Out    String String         -- Output of the computation
               | Target String String   -- Target that's not involved
               | Str String             -- String with trailing " "
               | NStr String            -- Just a string
               | ErrorMsg String        -- Error message: $(error x)
               | NL                     -- New line
```

```
                deriving (Show,Eq)
```

Each rule token can either be a string of some form, or a reference to a file. Note that for some file references, the tree is implicit: `Out` and `Target` files are always in the `BuildTree`.

Rules in Hake differ from plain Makefile rules in that they only consist of rule bodies (i.e., exactly what needs to be done), and the targets and dependencies are inferred (so they only need to be written once). An example may make this clear - here is a function which returns list of `RuleTokens` for maintaining a Unix library:

```
archive :: Options -> [String] -> String -> [ RuleToken ]
archive opts objs libpath =
    [ Str "ar cr ", Out arch libpath ]
    ++
    [ In BuildTree arch o | o <- objs ]
    ++
    [ NL, Str "ranlib ", Out arch libpath ]
```

The arguments to this function include a set of "options", which are used extensively inside Hake to pass around values like C flags, include paths, link options, etc., together with a set of object file paths and the path of a library file to build. The architecture "`arch`" is defined elsewhere (this example is from the file with rules specific to `x86_64`, so within the scope it is defined globally)

The library is referred to as an `Out` token, since it is a target of the rule, whereas the object files are referred to by `In` tokens, since they are prerequisites. Both are in the `arch` architecture, since they have presumably been built by other rules.

This function is called from another, `arch`-independent function called "`archiveLibrary`, which dispatches based on the architectures that need to be built for a given library. Hence, if a Hakefile at "`drivers/e1000/Hakefile`" contained the expression:

```
archiveLibrary "x86_64" "e1000drv" [ "e1000.o", "e1000srv.o"]
```

– the resulting Makefile would contain:

```
./x86_64/drivers/e1000/libe1000drv.a: \
                        ./x86_64/drivers/e1000/e1000.o \
                        ./x86_64/drivers/e1000/e1000srv.o
        ar cr ./x86_64/drivers/e1000/libe1000drv.a \
                        ./x86_64/drivers/e1000/e1000.o \
                        ./x86_64/drivers/e1000/e1000srv.o
        ranlib ./x86_64/drivers/e1000/libe1000drv.a
```

The precise definitions of each token are as follows:

In tokens are file references which are dependent inputs for a Make rule. In other words, they refer to files which will appear both in the rule body and the list of dependencies (the right hand side) in the rule head. **In** file references can be in any architecture.

Dep tokens are file references to implicit dependencies. In Make terms, these are file names which appear in the list of dependencies in rule head, but don't explicitly appear in the rule body.

PreDep tokens are like **Dep** tokens, but appear in the rule head following a | character. GNU Make will require these dependencies to be built only if they do not already exist - it does not check for modification times. In Barrelfish, such dependencies are used for files such as `errno.h` which must be generated first in order to calculate C dependencies, but which ultimately not all C files depend upon. Any true dependency of a C file on `errno.h` will be specified by the generated depend files, and thus override the **PreDep** declaration.

NoDep tokens are file references that are not dependencies at all. The file name only appears in the rule body, never in the head. For example, **NoDep** references are used for directories for include files.

Out tokens are file references to output files from a rule, which are mentioned in the rule body. This is the common case for most files generated by Make rules.

Target tokens are file references that are implicit outputs of the rule, but do not appear in the rule body. In Make terms they appear only in the left-hand side of the rule head, and not in the body.

Str tokens are simply strings. They will be followed in the Makefile by a space character, which is usually what you want.

NStr tokens are like **Str**, but not followed by a space. This is useful for situations like the `-I` flag to the C compiler, which takes a directory name (specified by a **NoDep** token) without any intervening whitespace.

ErrorMsg tokens are a way to incorporate error conditions into the Makefile - they are translated into the GNU make construct `$(error x)`.

NL tokens are simply newlines in the rule.

In practice, a Hakefile rarely has to resort to explicit `RuleTokens`, but instead calls functions inside Hake to return `HRules`.

3.3 Higher rule abstractions

The guts of Hake is mostly contained in the file `RuleDefs.hs`, which provides a big lattice of functions to automate generating rules for commonly used patterns. If you want to do more complex things than simply "`build application`" or "`build library`", it's a good idea to understand how these features use the definitions in `RuleDefs.hs`.

3.4 Target architectures

Most of the flexibility required of Hake in building for multiple architectures is simply coded into the Haskell source of the program.

For every target architecture (at time of writing, only `x86_64`), there is a file (`X64_64.hs`) which contains the definitions required to build the system for that target. Adding a new target architecture for Barrelfish involves writing a new one of these files (e.g. `ARM.hs`, or `X86_32.hs`, etc.) and modifying the code in `RuleDefs.hs` to dispatch to the correct module.

3.5 Host architectures

Hake at present supports only a single host architecture: the toolchain to build Barrelfish is specified once in the target architecture files (see above).

To add support for multiple host build environments, one way to slice the problem is for the target architecture modules to import different host architecture modules and decide which one to call to get tool and path defnitions at runtime.

3.6 Configuration

The file `hake/Config.hs` in the build directory contains all the configuration variables (at time of writing) used for Barrelfish. Unlike in Make or CMake, these are Haskell values of arbitrary type, since they are evaluated entirely within Hake.

To reconfigure a build tree, therefore, one modifies this file, and rebuilds Hake and the top-level Makefile. The `rehake` target performs this task.

Chapter 4

Bootstrapping and Configuring Hake

Hake is bootstrapped using a shell script found in the Barrelfish source tree in `hake/hake.sh`. This script is the place to start configuring a new core Barrelfish OS build tree, and must be run in the root of the new build directory.

`hake.sh` takes the following command-line options:

-s,–source-dir: This option is mandatory and specifies the path to the Barrelfish source directory tree.

-i,–install-dir: This option specifies a path to an alternative install directory, and defaults to `‘pwd‘`.

-a,–architecture: This option can be given multiple times and specifies the list of architectures to build Barrelfish for. Run the script with the `-h` option to get the default list of architectures.

-h,–help: Prints a usage message.

-n,–no-hake: This option simply rebuilds Hake, but does not run it to generate a Makefile. It can be handy for debugging Hake itself.

After parsing and checking arguments, `hake.sh` next creates a new configuration file `hake/Config.hs` in the build tree. The configuration options in this file are defaults: it is a copy of the template `hake/Config.hs.template` in the source tree.

If this file already exists in the build tree, however, it is left unchanged, which means that any user modifications to this file persist across multiple bootstrapping runs of `hake.sh`. If you really want to reconfigure a build tree from scratch, you should therefore remove everything in the build tree, including this file.

`hake.sh` next similarly creates a file called `symbolic_targets.mk` in the root of the build tree, if it does not already exist.

After this, Hake itself is recompiled in the build tree (including the new `Config.hs` file), and then run with default options (most of which will be picked up from `Config.hs`).

Chapter 5

Debugging Hakefiles

At least three things can go wrong when you modify or write a Hakefile.

5.1 The Hakefile has a compile error

If you make a mistake in a Hakefile, the most likely output you will see is a funny-looking Haskell compile error, e.g.:

```
../barrelfish.oothake/usr/pci/Hakefile:13:0:
    Couldn't match expected type `t -> [HRule]'
           against inferred type `[a]'
    In the expression:
        [build
           (application
              {target = "pci", flounderBindings = ["pci"],
               flounderDefs = ["monitor"],
               mackerelDevices = ["pci_hdr0", "pci_hdr1", ....],
 \ldots
 \ldots
<command line>: module is not loaded: `Hakefiles' (Hakefiles.hs)
```

Ignoring the last line for the moment, if you know enough Haskell this should tell you exactly what is wrong with some Hakefile. However, even if you don't know enough Haskell, it does say which Hakefile is at fault and whereabouts in the offending Hakefile the problem is (in this case line 13 of `usr/pci/Hakefile`).

Also, the file that Hake tried to compile will be left for you in `Hakefiles.hs`. If you look at this, you'll see it's constructed out of individual Hakefiles together with a preamble giving details of the files in the tree.

5.2 The Makefile has an error

Hake generates a single large Makefile at the top of the tree. While it's huge (often several 100,000 lines), it's actually very easy to understand since (a) it only refers to files, (b) it contains comments saying where each bit comes from, and (c) it barely uses any Make variables at all.

It is hard to persuade Hake to generate an invalid Makefile, but it's possible. If so, it may still be due to an error in some Hakefile, in which case look at the file comments preceding the line where Make thinks the error is to find out which Hakefile to look at.

The most common problem is actually due to out of date dependencies. Hake does its best to calculate dependencies properly, but sometimes (such as when Hakefiles themselves change) they get confused. In this case, the first thing to try to is completely remove the build tree and try again. As you get more of a feel for the system it's possible to more surgically remove bits of the tree (the Makefile knows how to recreate any part of the build tree).

5.3 The Makefile works, but the build fails

In this case, you've written valid Hake rules, but they don't do what you want them to. In this case as well, looking at the generated Makefile can often help work out what went wrong.

Chapter 6

Command-line arguments

The Hake binary built in a Barrelfish tree can be found in `/hake/hake`, and takes the following command-line arguments:

–source-dir: this option is mandatory and specifies the root of the source tree.

–output-filename: this option specifies the name of the output Makefile, and defaults to `Makefile`

–quiet: this option turns off some information and warning messages as Hake runs.

–verbose: this option increases the verbosity level of Hake's information messages.

–install-dir: this option specifies the install tree. It defaults to the build tree (the current working directory where Hake runs).

–architecture: this option can be specified multiple times and gives an architecture for Hake to build. It overrides the default list of architectures to build that was set when Hake was configured. At time of writing, supported architectures include `x86_64`, `x86_32`, `arm`, `arm11mp`, `beehive`, and `scc`.

6.0.1 Building an external application or library

Building an application or library *outside* the main Barrelfish tree involves invoking Hake directly (rather than bootstrapping with `hake.sh`), and requires to you have a pre-built Barrelfish tree with at least as many architectures built as you would like to build the application or library for.

For example, suppose `/projects/barrelfish/install` contains a core Barrelfish tree built for all supported architectures, and the user's home directory contains a small source tree `~/quake3` containing an application to be built for `x86_32` only. As long as this source tree has a correct Hakefile (or Hakefiles), the following should build the application:

```
$ mkdir quake_build
$ cd quake_build
$ /projects/barrelfish/install/hake/hake \
        --source-dir ~/quake \
        --install-dir /projects/barrelfish/install \
        --architecture x86_32
$ make -j 16
```

Chapter 7

Wishlist

Hake is missing many desirable features. Hopefully, this list will reduce in size over time. Here are a few:

- Support for multiple host build environments (such a Cygwin).
- The bootstrapping process for Hake, while short, is a little unsatisfactory.