Building and developing GHC

The GHC Team

Abstract

This Guide is primarily aimed at those who want to build and/or hack on GHC. It describes how to get started with building GHC on your machine, and how to tweak the settings to get the kind of build you want. It also describes the inner workings of the build system, so you can extend it, modify it, and use it to build your code.

The bulk of this guide applies to building on Unix systems; see Section 13, "Instructions for building under Windows" for Windows notes.

Table of Contents

1. Getting the sources

You can get your hands on the GHC sources in two ways:

Source distributions
: You have a supported platform, but (a) you like the warm fuzzy feeling of compiling things yourself; (b) you want to build something ‘‘extra”—e.g., a set of libraries with strictness-analysis turned off; or (c) you want to hack on GHC yourself.

  A source distribution contains complete sources for GHC. Not only that, but the more awkward machine-independent steps are done for you. For example, if you don't have **happy** you'll find it convenient that the source distribution contains the result of running **happy** on the parser specifications. If you don't want to alter the parser then this saves you having to find and install **happy**. You will still need a working version of GHC (version 5.x or later) on your machine in order to compile (most of) the sources, however.

The darcs repository.
: We make releases infrequently. If you want more up-to-the minute (but less tested) source code then you need to get access to our darcs repository.

  Information on accessing the darcs repository is on the wiki: http://hackage.haskell.org/trac/ghc/wiki/GhcDarcs.

The repository holds source code only. It holds no mechanically generated files at all. So if you check out a source tree from darcs you will need to install every utility so that you can build all the derived files from scratch.

2. Things to check before you start

Here's a list of things to check before you get started.

1. Disk space needed: from about 100Mb for a basic GHC build, up to probably 500Mb for a GHC build with everything included (libraries built several different ways, etc.).

2. Use an appropriate machine / operating system. Section 3, “What machines GHC runs on” lists the supported platforms; if yours isn't amongst these then you can try porting GHC (see Section 10, “Porting GHC”).

3. Be sure that the “pre-supposed” utilities are installed. Section 4, “Installing pre-supposed utilities” elaborates.

4. If you have any problem when building or installing the Glasgow tools, please check the “known pitfalls” (Section 11, “Known pitfalls in building Glasgow Haskell ”). Also check the FAQ for the version you're building, which is part of the User's Guide and available on the GHC web site [http://www.haskell.org/ghc/].

   If you feel there is still some shortcoming in our procedure or instructions, please report it.

   For GHC, please see the bug-reporting section of the GHC Users' Guide [http://www.haskell.org/ghc/docs/latest/set/bug-reporting.html], to maximise the usefulness of your report.

   If in doubt, please send a message to `<glasgow-haskell-bugs@haskell.org>`.

3. What machines GHC runs on

A “platform” is a architecture/manufacturer/operating-system combination, such as `sparc-sun-solaris2`. Other common ones are `alpha-dec-osf2`, `hppa1.1-hp-hpux9`, `i386-unknown-linux`, `i386-unknown-solaris2`, `i386-unknown-freebsd`, `i386-unknown-cygwin32`, `m68k-sun-sunos4`, `mips-sgi-irix5`, `sparc-sun-sunos4`, `sparc-sun-solaris2`, `powerpc-ibm-aix`.

Some libraries may only work on a limited number of platforms; for example, a sockets library is of no use unless the operating system supports the underlying BSDisms.

The GHC hierarchy of Porting Goodness: (a) Best is a native-code generator; (b) next best is a “registerised” port; (c) the bare minimum is an “unregisterised” port. (“Unregisterised” is so terrible that we won't say more about it).

Here's everything that's known about GHC ports. We identify platforms by their “canonical” CPU/Manufacturer/OS triple.

{osf,linux,freebsd,openbsd,netbsd} :	The OSF port is currently working (as of GHC version 5.02.1) and well supported. The native code generator is currently non-working. Other operating systems will require some minor porting.
sparc-sun-sunos4	Probably works with minor tweaks, hasn't been tested for a while.
sparc-sun-solaris2	Fully supported (at least for Solaris 2.7 and 2.6), including native-code generator.
sparc-unknown-openbsd	Supported, including native-code generator. The same should also be true of NetBSD
hppa1.1-hp-hpux (HP-PA boxes running HPUX 9.x)	A registerised port is available for version 4.08, but GHC hasn't been built on that platform since (as far as we know). No native-code generator.
i386-unknown-linux (PCs running Linux, ELF binary format)	GHC works registerised and has a native code generator. You *must* have GCC 2.7.x or later. NOTE about `glibc` versions: GHC binaries built on a system running `glibc 2.0` won't work on a system running `glibc 2.1`, and vice versa. In general, don't expect compatibility between `glibc` versions, even if the shared library version hasn't changed.
i386-unknown-freebsd (PCs running FreeBSD 2.2 or higher)	GHC works registerised. Pre-built packages are available in the native package format, so if you just need binaries you're better off just installing the package (it might even be on your installation CD!).
i386-unknown-openbsd (PCs running OpenBSD)	Supported, with native code generator. Packages are available through the ports system in the native package format.
i386-unknown-netbsd (PCs running NetBSD)	Will require some minor porting effort, but should work registerised.
i386-unknown-mingw32 (PCs running Windows)	Fully supported under Win9x, WinNT, Win2k, and WinXP. Includes a native code generator. Building from source requires a recent Cygwin [http://www.cygwin.com/] distribution to be installed.
ia64-unknown-linux	Supported, except there is no native code generator.
x86_64-unknown-linux , amd64-unknown-openbsd	Fully supported, with a native code generator and GHCi.
mips-sgi-irix5	Port has worked in the past, but hasn't been tested for some time (and will certainly have rotted in various ways). As usual, we don't have access to machines and there hasn't been an overwhelming demand for this port, but feel free to get in touch.
mips64-sgi-irix6	GHC currently works unregisterised.
powerpc-ibm-aix	Port currently doesn't work, needs some minimal porting effort. As usual, we don't have access to machines and there hasn't been an overwhelming demand for this port, but feel free to get in touch.
powerpc-apple-darwin	Supported registerised. Native code generator is almost working.
powerpc-apple-linux	Not supported (yet).

Various other systems have had GHC ported to them in the distant past, including various Motorola 68k boxes. The 68k support still remains, but porting to one of these systems will certainly be a non-trivial task.

4. Installing pre-supposed utilities

Here are the gory details about some utility programs you may need; **perl**, **gcc** and **happy** are the only important ones. (PVM is important if you're going for Parallel Haskell.) The **configure** script will tell you if you are missing something.

GHC
: GHC is required to build many of the tools, including GHC itself. If you need to port GHC to your platform because there isn't a binary distribution of GHC available, then see Section 10, “Porting GHC”.

  Which version of GHC you need will depend on the packages you intend to build. GHC itself will normally build using one of several older versions of itself - check the announcement or release notes for details.

Perl
: *You have to have Perl to proceed!* Perl version 5 at least is required. GHC has been known to tickle bugs in Perl, so if you find that Perl crashes when running GHC try updating (or downgrading) your Perl installation. Versions of Perl before 5.6 have been known to have various bugs tickled by GHC, so the configure script will look for version 5.6 or later.

  For Win32 platforms, you should use the binary supplied in the InstallShield (copy it to `/bin`). The Cygwin-supplied Perl seems not to work.

  Perl should be put somewhere so that it can be invoked by the `#!` script-invoking mechanism. The full pathname may need to be less than 32 characters long on some systems.

GNU C (**gcc**)
: Most GCC versions should work with the most recent GHC sources. Expect trouble if you use a recent GCC with an older GHC, though (trouble in the form of mis-compiled code, link errors, and errors from the `ghc-asm` script).

  If your GCC dies with “internal error” on some GHC source file, please let us know, so we can report it and get things improved. (Exception: on x86 boxes—you may need to fiddle with GHC's `-monly-N-regs` option; see the User's Guide)

GNU Make
: The fptools build system makes heavy use of features specific to GNU **make**, so you must have this installed in order to build any of the fptools suite.

  NB. it has been reported that version 3.79 no longer works to build GHC, and 3.80 is required.

Happy [http://www.haskell.org/happy]
: Happy is a parser generator tool for Haskell, and is used to generate GHC's parsers.

  If you start from a source tarball of GHC (i.e. not a darcs checkout), then you don't need Happy, because we supply the pre-processed versions of the Happy parsers. If you intend to modify the compiler and/or you're using a darcs checkout, then you need Happy.

Happy version 1.15 is currently required to build GHC.

Happy is written in Haskell, and is a project in the CVS repository (`fptools/happy`). It can be built from source, but bear in mind that you'll need GHC installed in order to build it. To avoid the chicken/egg problem, install a binary distribution of either Happy or GHC to get started. Happy distributions are available from Happy's Web Page [http://www.haskell.org/happy/].

Alex
: Alex is a lexical-analyser generator for Haskell, which GHC uses to generate its lexer.

Like Happy, you don't need Alex if you're building GHC from a source tarball, but you do need it if you're modifying GHC and/or building a darcs checkout.

Alex is written in Haskell and is a project in the darcs repository. Alex distributions are available from Alex's Web Page [http://www.haskell.org/alex/].

autoconf
: GNU autoconf is needed if you intend to build from the darcs sources, it is *not* needed if you just intend to build a standard source distribution.

Version 2.52 or later of the autoconf package is required. NB. version 2.13 will no longer work, as of GHC version 6.1.

autoreconf (from the autoconf package) recursively builds **configure** scripts from the corresponding `configure.ac` and `aclocal.m4` files. If you modify one of the latter files, you'll need **autoreconf** to rebuild the corresponding `configure`.

sed
: You need a working **sed** if you are going to build from sources. The build-configuration stuff needs it. GNU sed version 2.0.4 is no good! It has a bug in it that is tickled by the build-configuration. 2.0.5 is OK. Others are probably OK too (assuming we don't create too elaborate configure scripts.)

One `fptools` project is worth a quick note at this point, because it is useful for all the others: `glafp-utils` contains several utilities which aren't particularly Glasgow-ish, but Occasionally Indispensable. Like **lndir** for creating symbolic link trees.

4.1. Tools for building parallel GHC (GPH)

PVM version 3:
: PVM is the Parallel Virtual Machine on which Parallel Haskell programs run. (You only need this if you plan to run Parallel Haskell. Concurrent Haskell, which runs concurrent threads on a uniprocessor doesn't need it.) Underneath PVM, you can have (for example) a network of workstations (slow) or a multiprocessor box (faster).

The current version of PVM is 3.3.11; we use 3.3.7. It is readily available on the net; I think I got it from `research.att.com`, in `netlib`.

A PVM installation is slightly quirky, but easy to do. Just follow the `Readme` instructions.

bash: Sadly, the **gr2ps** script, used to convert “parallelism profiles” to PostScript, is written in Bash (GNU's Bourne Again shell). This bug will be fixed (someday).

4.2. Other useful tools

Flex: This is a quite-a-bit-better-than-Lex lexer. Used to build a couple of utilities in `glafp-utils`. Depending on your operating system, the supplied **lex** may or may not work; you should get the GNU version.

More tools are required if you want to format the documentation that comes with GHC and other fptools projects. See Section 9, “Building the documentation”.

5. Building from source

“I just want to build it!”

No problem. This recipe should build and install a working GHC with all the default settings. (unless you're on Windows, in which case go to Section 13, “Instructions for building under Windows”).

```
$ autoreconf1
$ ./configure
$ make
$ make install
```

For GHC, this will do a 2-stage bootstrap build of the compiler, with profiling libraries, and install the results in the default location (under `/usr/local` on Unix, for example).

The `configure` script is a standard GNU `autoconf` script, and accepts the usual options for changing install locations and the like. Run `./configure --help` for a list of options.

If you want to do anything at all non-standard, or you want to do some development, read on...

6. Quick start for GHC developers

This section is a copy of the file `ghc/HACKING` from the GHC source tree. It describes how to get started with setting up your build tree for developing GHC or its libraries, and how to start building.

```
Getting started with hacking on GHC
-----------------------------------

So you've decided to hack on GHC, congratulations!  We hope you have a
rewarding experience.  This file contains a few nuggets of information
that will help get you started right away, and point you in the
direction of more comprehensive documentation for later.


Setting up your build
---------------------
```

1 not necessary if you started from a source tarball

```
The GHC build tree is set up so that, by default, it builds a compiler
ready for installing and using.  That means full optimisation, and the
build can take a *long* time.  If you unpack your source tree and
right away say "./configure; make", expect to have to wait a while.

For hacking, you want the build to be quick - quick to build in the
first place, and quick to rebuild after making changes.  Tuning your
build setup can make the difference between several hours to build
GHC, and less than an hour.  Here's how to do it.

mk/build.mk is a GNU makefile that contains all your build settings.
By default, this file doesn't exist, and all the parameters are set to
their defaults in mk/config.mk (mk/config.mk is the place to look for
*all* the things you might want to tune).

A good mk/build.mk to start hacking on GHC is:

------
SRC_HC_OPTS     = -H32m -O -fasm -Rghc-timing
GhcStage1HcOpts = -O0 -DDEBUG
GhcLibHcOpts    = -O -fgenerics
GhcLibWays      =
SplitObjs       = NO
------

What do these options do?

SRC_HC_OPTS = -H32m -O -fasm -Rghc-timing

  These options are added to the command line for all Haskell
  compilations.  We turn on -fasm, because that halves compilation
  time at the expense of a few percent performance.  -Rghc-timing
  prints out a line of timing info about each compilation.  It's handy
  to keep an eye on.

GhcStage1HcOpts = -O0 -DDEBUG

  The options for building the stage1 compiler (these come after
  SRC_HC_OPTS, so you can override settings from there).  We turn off
  optimisation here, assuming you'll be modifying and testing stage1.
  With optimisation off, rebuilding GHC after modifying it will be
  *much* quicker, not only because the individual compilations will be
  quicker, but also there will be fewer dependencies between modules,
  so less stuff needs to be rebuilt after each modification.

  Also we turn on -DDEBUG, because that enables assertions and
  debugging code in the compiler itself.  Turning on DEBUG makes
  the compiler about 30% slower.

GhcLibHcOpts = -O -fgenerics

  You almost certainly want optimisation *on* when building
  libraries, otherwise the code you build with this compiler
  goes really slowly.  -fgenerics add generics support to the
  libraries - you can turn this off if you like (it'll make the
  libraries a bit smaller), but you won't be able to use Generics in
  the code you build against these libraries.

GhcLibWays =

  Normally the profiled libs are built.  Setting GhcLibWays to
  empty disables this, so you only build the normal libs.
```

```
SplitObjs = NO

  Object splitting causes each module to be split into smaller
  pieces in the final library, to reduce executable sizes when
  linking against the library.  It can be quite time and
  memory-consuming, so turn it off when you're hacking.


Actually building the bits
--------------------------

To just build everything, from the top level:

  $ autoreconf
  $ ./configure
  $ make
  $ make install


Building individual parts of the tree
-------------------------------------

The first thing to understand is that the source tree is built in two
passes.  First 'make boot' builds dependencies and any other tools
required as part of the build itself.  For example,
ghc/utils/genprimopcode is built as part of 'make boot', because it is
required to preprocess ghc/compiler/prelude/primops.txt.pp.

After 'make boot', 'make' will build everything.

If you say 'make' from the very top-level, the build system will
arrange to do the appropriate 'make boot' steps for you.  If you just
want to build in a subdirectory (eg. ghc), you have to do 'make boot'
yourself.  You don't need to 'make boot' after every single change,
but you might want to do it to update dependencies, for example.


Refining the setup
------------------

If you will be hacking mostly on libraries, then you probably want to
build stage1 with optimisation, because you're only building it once
but using it many times.

  GhcState1HcOpts = -O

If you are working on GHCi or Template Haskell, then you will be
building and modifying the stage 2 compiler.  Hence, you want to build
stage 1 with, and stage 2 without, optimisation.

  GhcState1HcOpts = -O
  GhcState2HcOpts = -O0 -DDEBUG

Take a look through mk/config.mk for more settings you might want to
override in build.mk.  Remember: don't modify config.mk directly (it
gets overwritten when you run ./configure).


Full optimisation
-----------------

To turn up everything to the max, for running performance tests for
example, try theses:
```

```
  SRC_HC_OPTS  = -H64m -O2
  GhcLibHcOpts = -O2
  SplitObjs    = YES

You can even add some more aggresive options, such as
-fliberate-case-threshold50, -funfolding-use-threshold50.


Roadmap
-------

A rough roadmap to the source tree:

 distrib        materials for building distributions

 docs           build system documentation

 ghc            The GHC Compiler
   rts          the runtime system and storage manager
   lib          libraries used in GHC and its tools
   utils tools that come with GHC, and tools used in the build
   compiler     the compiler itself
   driver       various scripts, and package databases
   docs         compiler documentation
   includes     header files shipped with GHC

 glafp-utils    tools for the build system

 libraries The hierarchical libraries

 nofib          A benchmark suite

 testsuite      The regression test suite


Resources
---------

The Building Guide

   Full documentation on the build system.
   http://www.haskell.org/ghc/docs/latest/html/building/index.html


The GHC Commentary

   Notes on the internals and architecture of GHC.  Much of this isn't
   up to date, but there is still lots of useful stuff in there.  Read
   in conjunction with the source code.
   http://www.cse.unsw.edu.au/~chak/haskell/ghc/comm/


Mailing lists

   Ask on glasgow-haskell-users@haskell.org if you have difficulties.
   If you're working with the current CVS sources of GHC, then
   cvs-ghc@haskell.org might be a more appropriate (developers hang
   out here).  See http://www.haskell.org/mailman/listinfo for
   subscription.


Happy Hacking!  --The GHC Team
```

7. Working with the build system

This rest of this guide is intended for duffers like me, who aren't really interested in Makefiles and systems configurations, but who need a mental model of the interlocking pieces so that they can make them work, extend them consistently when adding new software, and lay hands on them gently when they don't work.

7.1. Your source tree

The source code is held in your *source tree*. The root directory of your source tree *must* contain the following directories and files:

- `Makefile`: the root Makefile.
- `mk/`: the directory that contains the main Makefile code, shared by all the `fptools` software.
- `configure.ac`, `config.sub`, `config.guess`: these files support the configuration process.
- `install-sh`.

All the other directories are individual *projects* of the `fptools` system—for example, the Glasgow Haskell Compiler (`ghc`), the Happy parser generator (`happy`), the `nofib` benchmark suite, and so on. You can have zero or more of these. Needless to say, some of them are needed to build others.

The important thing to remember is that even if you want only one project (`happy`, say), you must have a source tree whose root directory contains `Makefile`, `mk/`, `configure.ac`, and the project(s) you want (`happy/` in this case). You cannot get by with just the `happy/` directory.

7.2. Build trees

If you just want to build the software once on a single platform, then your source tree can also be your build tree, and you can skip the rest of this section.

We often want to build multiple versions of our software for different architectures, or with different options (e.g. profiling). It's very desirable to share a single copy of the source code among all these builds.

So for every source tree we have zero or more *build trees*. Each build tree is initially an exact copy of the source tree, except that each file is a symbolic link to the source file, rather than being a copy of the source file. There are "standard" Unix utilities that make such copies, so standard that they go by different names: **lndir**, **mkshadowdir** are two (If you don't have either, the source distribution includes sources for the X11 **lndir**—check out `fptools/glafp-utils/lndir`). See Section 7.4, "The story so far" for a typical invocation.

The build tree does not need to be anywhere near the source tree in the file system. Indeed, one advantage of separating the build tree from the source is that the build tree can be placed in a non-backed-up partition, saving your systems support people from backing up untold megabytes of easily-regenerated, and rapidly-changing, gubbins. The golden rule is that (with a single exception—Section 7.3, "Getting the build you want") *absolutely everything in the build tree is either a symbolic link to the source tree, or else is mechanically generated*. It should be perfectly OK for your build tree to vanish overnight; an hour or two compiling and you're on the road again.

You need to be a bit careful, though, that any new files you create (if you do any development work) are in the source tree, not a build tree!

Remember, that the source files in the build tree are *symbolic links* to the files in the source tree. (The build tree soon accumulates lots of built files like `Foo.o`, as well.) You can *delete* a source file from the build tree without affecting the source tree (though it's an odd thing to do). On the other hand, if you *edit* a source file from the build tree, you'll edit the source-tree file directly. (You can set up Emacs so that if you edit a source file from the build tree, Emacs will silently create an edited copy of the source file in the build tree, leaving the source file unchanged; but the danger is that you think you've edited the source file whereas actually all you've done is edit the build-tree copy. More commonly you do want to edit the source file.)

Like the source tree, the top level of your build tree must be (a linked copy of) the root directory of the `fptools` suite. Inside Makefiles, the root of your build tree is called `$(FPTOOLS_TOP)`. In the rest of this document path names are relative to `$(FPTOOLS_TOP)` unless otherwise stated. For example, the file `ghc/mk/target.mk` is actually `$(FPTOOLS_TOP)/ghc/mk/target.mk`.

7.3. Getting the build you want

When you build `fptools` you will be compiling code on a particular *host platform*, to run on a particular *target platform* (usually the same as the host platform). The difficulty is that there are minor differences between different platforms; minor, but enough that the code needs to be a bit different for each. There are some big differences too: for a different architecture we need to build GHC with a different native-code generator.

There are also knobs you can turn to control how the `fptools` software is built. For example, you might want to build GHC optimised (so that it runs fast) or unoptimised (so that you can compile it fast after you've modified it. Or, you might want to compile it with debugging on (so that extra consistency-checking code gets included) or off. And so on.

All of this stuff is called the *configuration* of your build. You set the configuration using a three-step process.

Step 1: get ready for configuration.

NOTE: if you're starting from a source distribution, rather than darcs sources, you can skip this step.

Change directory to `$(FPTOOLS_TOP)` and issue the command

```
$ autoreconf
```

(with no arguments). This GNU program (recursively) converts `$(FPTOOLS_TOP)/configure.ac` and `$(FPTOOLS_TOP)/aclocal.m4` to a shell script called `$(FPTOOLS_TOP)/configure`. If **autoreconf** bleats that it can't write the file `configure`, then delete the latter and try again. Note that you must use **autoreconf**, and not the old **autoconf**! If you erroneously use the latter, you'll get a message like "No rule to make target 'mk/config.h.in'".

Some projects, including GHC, have their own configure script. **autoreconf** takes care of that, too, so all you have to do is calling **autoreconf** in the top-level directory `$(FPTOOLS_TOP)`.

These steps are completely platform-independent; they just mean that the human-written files (`configure.ac` and `aclocal.m4`) can be short, although the resulting files (the **configure** shell scripts and the C header template `mk/config.h.in`) are long.

Step 2: system configuration.

Runs the newly-created **configure** script, thus:

```
$ ./configure [args]
```

configure's mission is to scurry round your computer working out what architecture it has, what operating system, whether it has the `vfork` system call, where **tar** is kept, whether **gcc** is available, where various obscure `#include` files are, whether it's a leap year, and what the systems manager had for lunch. It communicates these snippets of information in two ways:

- It translates `mk/config.mk.in` to `mk/config.mk`, substituting for things between "`@`" brackets. So, "`@HaveGcc@`" will be replaced by "`YES`" or "`NO`" depending on what **configure** finds. `mk/config.mk` is included by every Makefile (directly or indirectly), so the configuration information is thereby communicated to all Makefiles.

- It translates `mk/config.h.in` to `mk/config.h`. The latter is `#included` by various C programs, which can thereby make use of configuration information.

configure takes some optional arguments. Use `./configure --help` to get a list of the available arguments. Here are some of the ones you might need:

`--with-ghc=`*`path`*
: Specifies the path to an installed GHC which you would like to use. This compiler will be used for compiling GHC-specific code (eg. GHC itself). This option *cannot* be specified using `build.mk` (see later), because **configure** needs to auto-detect the version of GHC you're using. The default is to look for a compiler named `ghc` in your path.

`--with-hc=`*`path`*
: Specifies the path to any installed Haskell compiler. This compiler will be used for compiling generic Haskell code. The default is to use `ghc`.

`--with-gcc=`*`path`*
: Specifies the path to the installed GCC. This compiler will be used to compile all C files, *except* any generated by the installed Haskell compiler, which will have its own idea of which C compiler (if any) to use. The default is to use `gcc`.

Step 3: build configuration.
: Next, you say how this build of `fptools` is to differ from the standard defaults by creating a new file `mk/build.mk` *in the build tree*. This file is the one and only file you edit in the build tree, precisely because it says how this build differs from the source. (Just in case your build tree does die, you might want to keep a private directory of `build.mk` files, and use a symbolic link in each build tree to point to the appropriate one.) So `mk/build.mk` never exists in the source tree—you create one in each build tree from the template. We'll discuss what to put in it shortly.

And that's it for configuration. Simple, eh?

What do you put in your build-specific configuration file `mk/build.mk`? *For almost all purposes all you will do is put make variable definitions that override those in* `mk/config.mk.in`. The whole point of `mk/config.mk.in`—and its derived counterpart `mk/config.mk`—is to define the build configuration. It is heavily commented, as you will see if you look at it. So generally, what you do is look at `mk/config.mk.in`, and add definitions in `mk/build.mk` that override any of the `config.mk` definitions that you want to change. (The override occurs because the main boilerplate file, `mk/boilerplate.mk`, includes `build.mk` after `config.mk`.)

For your convenience, there's a file called `build.mk.sample` that can serve as a starting point for your `build.mk`.

For example, `config.mk.in` contains the definition:

```
GhcHcOpts=-O -Rghc-timing
```

The accompanying comment explains that this is the list of flags passed to GHC when building GHC itself. For doing development, it is wise to add `-DDEBUG`, to enable debugging code. So you would add the following to `build.mk`:

or, if you prefer,

```
GhcHcOpts += -DDEBUG
```

GNU **make** allows existing definitions to have new text appended using the “+=” operator, which is quite a convenient feature.)

If you want to remove the `-O` as well (a good idea when developing, because the turn-around cycle gets a lot quicker), you can just override `GhcLibHcOpts` altogether:

```
GhcHcOpts=-DDEBUG -Rghc-timing
```

When reading `config.mk.in`, remember that anything between “@...@” signs is going to be substituted by **configure** later. You *can* override the resulting definition if you want, but you need to be a bit surer what you are doing. For example, there's a line that says:

```
TAR = @TarCmd@
```

This defines the Make variables `TAR` to the pathname for a **tar** that **configure** finds somewhere. If you have your own pet **tar** you want to use instead, that's fine. Just add this line to `mk/build.mk`:

```
TAR = mytar
```

You do not *have* to have a `mk/build.mk` file at all; if you don't, you'll get all the default settings from `mk/config.mk.in`.

You can also use `build.mk` to override anything that **configure** got wrong. One place where this happens often is with the definition of `FPTOOLS_TOP_ABS`: this variable is supposed to be the canonical path to the top of your source tree, but if your system uses an automounter then the correct directory is hard to find automatically. If you find that **configure** has got it wrong, just put the correct definition in `build.mk`.

7.4. The story so far

Let's summarise the steps you need to carry to get yourself a fully-configured build tree from scratch.

1. Get your source tree from somewhere (darcs repository or source distribution). Say you call the root directory `myfptools` (it does not have to be called `fptools`). Make sure that you have the essential files (see Section 7.1, "Your source tree").

2. (Optional) Use **lndir** or **mkshadowdir** to create a build tree.

   ```
   $ cd myfptools
   $ mkshadowdir . /scratch/joe-bloggs/myfptools-sun4
   ```

   (N.B. **mkshadowdir**'s first argument is taken relative to its second.) You probably want to give the build tree a name that suggests its main defining characteristic (in your mind at least), in case you later add others.

3. Change directory to the build tree. Everything is going to happen there now.

   ```
   $ cd /scratch/joe-bloggs/myfptools-sun4
   ```

4. Prepare for system configuration:

   ```
   $ autoreconf
   ```

   (You can skip this step if you are starting from a source distribution, and you already have `configure` and `mk/config.h.in`.)

5. Do system configuration:

   ```
   $ ./configure
   ```

   Don't forget to check whether you need to add any arguments to `configure`; for example, a common requirement is to specify which GHC to use with `--with-ghc=`*`ghc`*.

6. Create the file `mk/build.mk`, adding definitions for your desired configuration options.

   ```
   $ emacs mk/build.mk
   ```

You can make subsequent changes to `mk/build.mk` as often as you like. You do not have to run any further configuration programs to make these changes take effect. In theory you should, however, say **gmake clean**, **gmake all**, because configuration option changes could affect anything—but in practice you are likely to know what's affected.

7.5. Making things

At this point you have made yourself a fully-configured build tree, so you are ready to start building real things.

The first thing you need to know is that *you must use GNU **make**, usually called **gmake**, not standard Unix **make***. If you use standard Unix **make** you will get all sorts of error messages (but no damage) because the `fptools` **Makefiles** use GNU **make**'s facilities extensively.

To just build the whole thing, **cd** to the top of your `fptools` tree and type **gmake**. This will prepare the tree and build the various projects in the correct order.

7.6. Bootstrapping GHC

GHC requires a 2-stage bootstrap in order to provide full functionality, including GHCi. By a 2-stage bootstrap, we mean that the compiler is built once using the installed GHC, and then again using the compiler built in the first stage. You can also build a stage 3 compiler, but this normally isn't necessary except to verify that the stage 2 compiler is working properly.

Note that when doing a bootstrap, the stage 1 compiler must be built, followed by the runtime system and libraries, and then the stage 2 compiler. The correct ordering is implemented by the top-level fptools `Makefile`, so if you want everything to work automatically it's best to start **make** from the top of the tree. When building GHC, the top-level fptools `Makefile` is set up to do a 2-stage bootstrap by default (when you say **make**). Some other targets it supports are:

stage1
: Build everything as normal, including the stage 1 compiler.

stage2
: Build the stage 2 compiler only.

stage3
: Build the stage 3 compiler only.

bootstrap, bootstrap2
: Build stage 1 followed by stage 2.

bootstrap3
: Build stages 1, 2 and 3.

install
: Install everything, including the compiler built in stage 2. To override the stage, say `make install stage=`*n* where *n* is the stage to install.

The top-level `Makefile` also arranges to do the appropriate `make boot` steps (see below) before actually building anything.

The `stage1`, `stage2` and `stage3` targets also work in the `ghc/compiler` directory, but don't forget that each stage requires its own `make boot` step: for example, you must do

```
$ make boot stage=2
```

before `make stage2` in `ghc/compiler`.

7.7. Standard Targets

In any directory you should be able to make the following:

`boot`
: does the one-off preparation required to get ready for the real work. Notably, it does **gmake depend** in all directories that contain programs. It also builds the necessary tools for compilation to proceed.

: Invoking the `boot` target explicitly is not normally necessary. From the top-level `fptools` directory, invoking `gmake` causes `gmake boot all` to be invoked in each of the project subdirectories, in the order specified by `$(AllTargets)` in `config.mk`.

: If you're working in a subdirectory somewhere and need to update the dependencies, `gmake boot` is a good way to do it.

`all`
: makes all the final target(s) for this Makefile. Depending on which directory you are in a “final target” may be an executable program, a library archive, a shell script, or a Postscript file. Typing **gmake** alone is generally the same as

typing **gmake all**.

`install`
: installs the things built by `all` (except for the documentation). Where does it install them? That is specified by `mk/config.mk.in`; you can override it in `mk/build.mk`, or by running **configure** with command-line arguments like `--bindir=/home/simonpj/bin`; see `./configure --help` for the full details.

`install-docs`
: installs the documentation. Otherwise behaves just like `install`.

`uninstall`
: reverses the effect of `install`.

`clean`
: Delete all files from the current directory that are normally created by building the program. Don't delete the files that record the configuration, or files generated by **gmake boot**. Also preserve files that could be made by building, but normally aren't because the distribution comes with them.

`distclean`
: Delete all files from the current directory that are created by configuring or building the program. If you have unpacked the source and built the program without creating any other files, `make distclean` should leave only the files that were in the distribution.

`mostlyclean`
: Like `clean`, but may refrain from deleting a few files that people normally don't want to recompile.

`maintainer-clean`
: Delete everything from the current directory that can be reconstructed with this Makefile. This typically includes everything deleted by `distclean`, plus more: C source files produced by Bison, tags tables, Info files, and so on.

  One exception, however: `make maintainer-clean` should not delete `configure` even if `configure` can be remade using a rule in the `Makefile`. More generally, `make maintainer-clean` should not delete anything that needs to exist in order to run `configure` and then begin to build the program.

`check`
: run the test suite.

All of these standard targets automatically recurse into sub-directories. Certain other standard targets do not:

`configure`
: is only available in the root directory `$(FPTOOLS_TOP)`; it has been discussed in Section 7.3, “Getting the build you want”.

`depend`
: make a `.depend` file in each directory that needs it. This `.depend` file contains mechanically-generated dependency information; for example, suppose a directory contains a Haskell source module `Foo.lhs` which imports another module `Baz`. Then the generated `.depend` file will contain the dependency:

```
Foo.o : Baz.hi
```

  which says that the object file `Foo.o` depends on the interface file `Baz.hi` generated by compiling module `Baz`. The `.depend` file is automatically included by every Makefile.

`binary-dist`
: make a binary distribution. This is the target we use to build the binary distributions of GHC and Happy.

`dist` make a source distribution. Note that this target does “make distclean” as part of its work; don't use it if you want to keep what you've built.

Most `Makefiles` have targets other than these. You can discover them by looking in the `Makefile` itself.

7.8. Using a project from the build tree

If you want to build GHC (say) and just use it direct from the build tree without doing `make install` first, you can run the in-place driver script: `ghc/compiler/ghc-inplace`.

Do *NOT* use `ghc/compiler/ghc`, or `ghc/compiler/ghc-6.xx`, as these are the scripts intended for installation, and contain hard-wired paths to the installed libraries, rather than the libraries in the build tree.

Happy can similarly be run from the build tree, using `happy/src/happy-inplace`, and similarly for Alex and Haddock.

7.9. Fast Making

Sometimes the dependencies get in the way: if you've made a small change to one file, and you're absolutely sure that it won't affect anything else, but you know that **make** is going to rebuild everything anyway, the following hack may be useful:

```
$ gmake FAST=YES
```

This tells the make system to ignore dependencies and just build what you tell it to. In other words, it's equivalent to temporarily removing the `.depend` file in the current directory (where **mkdependHS** and friends store their dependency information).

A bit of history: GHC used to come with a **fastmake** script that did the above job, but GNU make provides the features we need to do it without resorting to a script. Also, we've found that fastmaking is less useful since the advent of GHC's recompilation checker (see the User's Guide section on "Separate Compilation").

8. The `Makefile` architecture

make is great if everything works—you type **gmake install** and lo! the right things get compiled and installed in the right places. Our goal is to make this happen often, but somehow it often doesn't; instead some weird error message eventually emerges from the bowels of a directory you didn't know existed.

The purpose of this section is to give you a road-map to help you figure out what is going right and what is going wrong.

8.1. Debugging

Debugging `Makefiles` is something of a black art, but here's a couple of tricks that we find particularly useful. The following command allows you to see the contents of any make variable in the context of the current `Makefile`:

```
$ make show VALUE=HS_SRCS
```

where you can replace `HS_SRCS` with the name of any variable you wish to see the value of.

GNU make has a `-d` option which generates a dump of the decision procedure used to arrive at a conclusion about which files should be recompiled. Sometimes useful for tracking down problems with superfluous or missing recompilations.

8.2. A small project

To get started, let us look at the `Makefile` for an imaginary small `fptools` project, `small`. Each project in `fptools` has its own directory in `FPTOOLS_TOP`, so the `small` project will have its own directory `FPOOLS_TOP/small/`. Inside the `small/` directory there will be a `Makefile`, looking something like this:

```
# Makefile for fptools project "small"

TOP = ..
include $(TOP)/mk/boilerplate.mk

SRCS = $(wildcard *.lhs) $(wildcard *.c)
HS_PROG = small

include $(TOP)/target.mk
```

this `Makefile` has three sections:

1. The first section includes [2] a file of “boilerplate” code from the level above (which in this case will be `FPTOOLS_TOP/mk/boilerplate.mk`). As its name suggests, `boilerplate.mk` consists of a large quantity of standard `Makefile` code. We discuss this boilerplate in more detail in Section 8.5, “The main `mk/boilerplate.mk` file”.

   Before the `include` statement, you must define the **make** variable `TOP` to be the directory containing the `mk` directory in which the `boilerplate.mk` file is. It is *not* OK to simply say

   ```
   include ../mk/boilerplate.mk   # NO NO NO
   ```

   Why? Because the `boilerplate.mk` file needs to know where it is, so that it can, in turn, `include` other files. (Unfortunately, when an `include`d file does an `include`, the filename is treated relative to the directory in which **gmake** is being run, not the directory in which the `included` sits.) In general, *every file `foo.mk` assumes that `$(TOP)/mk/foo.mk` refers to itself.* It is up to the `Makefile` doing the `include` to ensure this is the case.

   Files intended for inclusion in other `Makefile`s are written to have the following property: *after `foo.mk` is `include`d, it leaves `TOP` containing the same value as it had just before the `include` statement*. In our example, this invariant guarantees that the `include` for `target.mk` will look in the same directory as that for `boilerplate.mk`.

2. The second section defines the following standard **make** variables: `SRCS` (the source files from which is to be built), and `HS_PROG` (the executable binary to be built). We will discuss in more detail what the “standard variables” are, and how they affect what happens, in Section 8.8, “The main `mk/target.mk` file”.

   The definition for `SRCS` uses the useful GNU **make** construct `$(wildcard $pat$)`, which expands to a list of all the files matching the pattern `pat` in the current directory. In this example, `SRCS` is set to the list of all the `.lhs` and `.c` files in the directory. (Let's suppose there is one of each, `Foo.lhs` and `Baz.c`.)

3. The last section includes a second file of standard code, called `target.mk`. It contains the rules that tell **gmake** how to make the standard targets (Section 7.7, “Standard Targets”). Why, you ask,

[2] One of the most important features of GNU **make** that we use is the ability for a `Makefile` to include another named file, very like cpp's `#include` directive.

can't this standard code be part of `boilerplate.mk`? Good question. We discuss the reason later, in Section 8.4, "Boilerplate architecture".

You do not *have* to `include` the `target.mk` file. Instead, you can write rules of your own for all the standard targets. Usually, though, you will find quite a big payoff from using the canned rules in `target.mk`; the price tag is that you have to understand what canned rules get enabled, and what they do (Section 8.8, "The main `mk/target.mk` file").

In our example `Makefile`, most of the work is done by the two `include`d files. When you say **gmake all**, the following things happen:

- **gmake** figures out that the object files are `Foo.o` and `Baz.o`.
- It uses a boilerplate pattern rule to compile `Foo.lhs` to `Foo.o` using a Haskell compiler. (Which one? That is set in the build configuration.)
- It uses another standard pattern rule to compile `Baz.c` to `Baz.o`, using a C compiler. (Ditto.)
- It links the resulting `.o` files together to make `small`, using the Haskell compiler to do the link step. (Why not use **ld**? Because the Haskell compiler knows what standard libraries to link in. How did **gmake** know to use the Haskell compiler to do the link, rather than the C compiler? Because we set the variable `HS_PROG` rather than `C_PROG`.)

All `Makefile`s should follow the above three-section format.

8.3. A larger project

Larger projects are usually structured into a number of sub-directories, each of which has its own `Makefile`. (In very large projects, this sub-structure might be iterated recursively, though that is rare.) To give you the idea, here's part of the directory structure for the (rather large) GHC project:

```
$(FPTOOLS_TOP)/ghc/
  Makefile
  mk/
    boilerplate.mk
    rules.mk
   docs/
    Makefile
    ...source files for documentation...
   driver/
    Makefile
    ...source files for driver...
   compiler/
    Makefile
    parser/...source files for parser...
    renamer/...source files for renamer...
    ...etc...
```

The sub-directories `docs`, `driver`, `compiler`, and so on, each contains a sub-component of GHC, and each has its own `Makefile`. There must also be a `Makefile` in `$(FPTOOLS_TOP)/ghc`. It does most of its work by recursively invoking **gmake** on the `Makefile`s in the sub-directories. We say that `ghc/Makefile` is a *non-leaf `Makefile`*, because it does little except organise its children, while the `Makefile`s in the sub-directories are all *leaf `Makefile`s*. (In principle the sub-directories might themselves contain a non-leaf `Makefile` and several sub-sub-directories, but that does not happen in GHC.)

The `Makefile` in `ghc/compiler` is considered a leaf `Makefile` even though the `ghc/compiler` has sub-directories, because these sub-directories do not themselves have `Makefiles` in them. They are just used to structure the collection of modules that make up GHC, but all are managed by the single `Makefile` in `ghc/compiler`.

You will notice that `ghc/` also contains a directory `ghc/mk/`. It contains GHC-specific `Makefile` boilerplate code. More precisely:

- `ghc/mk/boilerplate.mk` is included at the top of `ghc/Makefile`, and of all the leaf `Makefiles` in the sub-directories. It in turn `includes` the main boilerplate file `mk/boilerplate.mk`.
- `ghc/mk/target.mk` is `include`d at the bottom of `ghc/Makefile`, and of all the leaf `Makefiles` in the sub-directories. It in turn `includes` the file `mk/target.mk`.

So these two files are the place to look for GHC-wide customisation of the standard boilerplate.

8.4. Boilerplate architecture

Every `Makefile` includes a `boilerplate.mk` file at the top, and `target.mk` file at the bottom. In this section we discuss what is in these files, and why there have to be two of them. In general:

- `boilerplate.mk` consists of:

  - *Definitions of millions of **make** variables* that collectively specify the build configuration. Examples: `HC_OPTS`, the options to feed to the Haskell compiler; `NoFibSubDirs`, the sub-directories to enable within the `nofib` project; `GhcWithHc`, the name of the Haskell compiler to use when compiling GHC in the `ghc` project.
  - *Standard pattern rules* that tell **gmake** how to construct one file from another.

  `boilerplate.mk` needs to be `include`d at the *top* of each `Makefile`, so that the user can replace the boilerplate definitions or pattern rules by simply giving a new definition or pattern rule in the `Makefile`. **gmake** simply takes the last definition as the definitive one.

  Instead of *replacing* boilerplate definitions, it is also quite common to *augment* them. For example, a `Makefile` might say:

  ```
  SRC_HC_OPTS += -O
  ```

  thereby adding “`-O`” to the end of `SRC_HC_OPTS`.

- `target.mk` contains **make** rules for the standard targets described in Section 7.7, “Standard Targets”. These rules are selectively included, depending on the setting of certain **make** variables. These variables are usually set in the middle section of the `Makefile` between the two `includes`.

  `target.mk` must be included at the end (rather than being part of `boilerplate.mk`) for several tiresome reasons:

  - **gmake** commits target and dependency lists earlier than it should. For example, `target.mk` has a rule that looks like this:

    ```
    $(HS_PROG) : $(OBJS)
          $(HC) $(LD_OPTS) $< -o $@
    ```

If this rule was in `boilerplate.mk` then `$(HS_PROG)` and `$(OBJS)` would not have their final values at the moment **gmake** encountered the rule. Alas, **gmake** takes a snapshot of their current values, and wires that snapshot into the rule. (In contrast, the commands executed when the rule "fires" are only substituted at the moment of firing.) So, the rule must follow the definitions given in the `Makefile` itself.

- Unlike pattern rules, ordinary rules cannot be overriden or replaced by subsequent rules for the same target (at least, not without an error message). Including ordinary rules in `boilerplate.mk` would prevent the user from writing rules for specific targets in specific cases.
- There are a couple of other reasons I've forgotten, but it doesn't matter too much.

8.5. The main `mk/boilerplate.mk` file

If you look at `$(FPTOOLS_TOP)/mk/boilerplate.mk` you will find that it consists of the following sections, each held in a separate file:

`config.mk`
: is the build configuration file we discussed at length in Section 7.3, "Getting the build you want".

`paths.mk`
: defines **make** variables for pathnames and file lists. This file contains code for automatically compiling lists of source files and deriving lists of object files from those. The results can be overriden in the `Makefile`, but in most cases the automatic setup should do the right thing.

  The following variables may be set in the `Makefile` to affect how the automatic source file search is done:

  `ALL_DIRS`
  : Set to a list of directories to search in addition to the current directory for source files.

  `EXCLUDED_SRCS`
  : Set to a list of source files (relative to the current directory) to omit from the automatic search. The source searching machinery is clever enough to know that if you exclude a source file from which other sources are derived, then the derived sources should also be excluded. For example, if you set `EXCLUDED_SRCS` to include `Foo.y`, then `Foo.hs` will also be excluded.

  `EXTRA_SRCS`
  : Set to a list of extra source files (perhaps in directories not listed in `ALL_DIRS`) that should be considered.

  The results of the automatic source file search are placed in the following make variables:

  `SRCS`
  : All source files found, sorted and without duplicates, including those which might not exist yet but will be derived from other existing sources. `SRCS` *can* be overriden if necessary, in which case the variables below will follow suit.

  `HS_SRCS`
  : all Haskell source files in the current directory, including those derived from other source files (eg. Happy sources also give rise to Haskell sources).

  `HS_OBJS`
  : Object files derived from `HS_SRCS`.

`HS_IFACES`	Interface files (`.hi` files) derived from `HS_SRCS`.
`C_SRCS`	All C source files found.
`C_OBJS`	Object files derived from `C_SRCS`.
`SCRIPT_SRCS`	All script source files found (`.lprl` files).
`SCRIPT_OBJS`	"object" files derived from `SCRIPT_SRCS` (`.prl` files).
`HSC_SRCS`	All `hsc2hs` source files (`.hsc` files).
`HAPPY_SRCS`	All `happy` source files (`.y` or `.hy` files).
`OBJS`	the concatenation of `$(HS_OBJS)`, `$(C_OBJS)`, and `$(SCRIPT_OBJS)`.

Any or all of these definitions can easily be overriden by giving new definitions in your `Makefile`.

What, exactly, does `paths.mk` consider a "source file" to be? It's based on the file's suffix (e.g. `.hs`, `.lhs`, `.c`, `.hy`, etc), but this is the kind of detail that changes, so rather than enumerate the source suffices here the best thing to do is to look in `paths.mk`.

`opts.mk` defines **make** variables for option strings to pass to each program. For example, it defines `HC_OPTS`, the option strings to pass to the Haskell compiler. See Section 8.7, "Pattern rules and options".

`suffix.mk` defines standard pattern rules—see Section 8.7, "Pattern rules and options".

Any of the variables and pattern rules defined by the boilerplate file can easily be overridden in any particular `Makefile`, because the boilerplate `include` comes first. Definitions after this `include` directive simply override the default ones in `boilerplate.mk`.

8.6. Platform settings

There are three platforms of interest when building GHC:

The *build* platform
: The platform on which we are doing this build.

The *host* platform
: The platform on which these binaries will run.

The *target* platform
: The platform for which this compiler will generate code.

These platforms are set when running the `configure` script, using the `--build`, `--host`, and `--target` options. The `mk/config.mk` file defines several symbols related to the platform settings (see `mk/config.mk` for details).

We don't currently support build & host being different, because the build process creates binaries that are both run during the build, and also installed.

If host and target are different, then we are building a cross-compiler. For GHC, this means a compiler which will generate intermediate .hc files to port to the target architecture for bootstrapping. The libraries and stage 2 compiler will be built as HC files for the target system (see Section 10, "Porting GHC" for details.

More details on when to use BUILD, HOST or TARGET can be found in the comments in `config.mk`.

8.7. Pattern rules and options

The file `suffix.mk` defines standard *pattern rules* that say how to build one kind of file from another, for example, how to build a `.o` file from a `.c` file. (GNU **make**'s *pattern rules* are more powerful and easier to use than Unix **make**'s *suffix rules*.)

Almost all the rules look something like this:

```
%.o : %.c
       $(RM) $@
       $(CC) $(CC_OPTS) -c $< -o $@
```

Here's how to understand the rule. It says that *something*`.o` (say `Foo.o`) can be built from *something*`.c` (`Foo.c`), by invoking the C compiler (path name held in `$(CC)`), passing to it the options `$(CC_OPTS)` and the rule's dependent file of the rule `$<` (`Foo.c` in this case), and putting the result in the rule's target `$@` (`Foo.o` in this case).

Every program is held in a **make** variable defined in `mk/config.mk`—look in `mk/config.mk` for the complete list. One important one is the Haskell compiler, which is called `$(HC)`.

Every program's options are are held in a **make** variables called `<prog>_OPTS`. the `<prog>_OPTS` variables are defined in `mk/opts.mk`. Almost all of them are defined like this:

```
CC_OPTS = \
  $(SRC_CC_OPTS) $(WAY$(_way)_CC_OPTS) $($*_CC_OPTS) $(EXTRA_CC_OPTS)
```

The four variables from which `CC_OPTS` is built have the following meaning:

`SRC_CC_OPTS`: options passed to all C compilations.

: options passed to C compilations for way `<way>`. For example, `WAY_mp_CC_OPTS` gives options to pass to the C compiler when compiling way `mp`. The variable `WAY_CC_OPTS` holds options to pass to the C compiler when compiling the standard way. (Section 8.10, “Way management” dicusses multi-way compilation.)

`<module>_CC_OPTS`: options to pass to the C compiler that are specific to module `<module>`. For example, `SMap_CC_OPTS` gives the specific options to pass to the C compiler when compiling `SMap.c`.

`EXTRA_CC_OPTS`: extra options to pass to all C compilations. This is intended for command line use, thus:

```
$ gmake libHS.a EXTRA_CC_OPTS="-v"
```

8.8. The main `mk/target.mk` file

`target.mk` contains canned rules for all the standard targets described in Section 7.7, “Standard Targets”. It is complicated by the fact that you don't want all of these rules to be active in every `Makefile`. Rather than have a plethora of tiny files which you can include selectively, there is a single file, `target.mk`, which selectively includes rules based on whether you have defined certain variables in your `Makefile`. This section explains what rules you get, what variables control them, and what the rules do. Hopefully, you will also get enough of an idea of what is supposed to happen that you can read and understand any weird special cases yourself.

`HS_PROG`. If `HS_PROG` is defined, you get rules with the following targets:

- `HS_PROG` itself. This rule links `$(OBJS)` with the Haskell runtime system to get an executable called `$(HS_PROG)`.
- `install` installs `$(HS_PROG)` in `$(bindir)`.

`C_PROG` is similar to `HS_PROG`, except that the link step links `$(C_OBJS)` with the C runtime system.

`LIBRARY` is similar to `HS_PROG`, except that it links `$(LIB_OBJS)` to make the library archive `$(LIBRARY)`, and `install` installs it in `$(libdir)`.

`LIB_DATA` …

`LIB_EXEC` …

`HS_SRCS`, `C_SRCS`. If `HS_SRCS` is defined and non-empty, a rule for the target `depend` is included, which generates dependency information for Haskell programs. Similarly for `C_SRCS`.

All of these rules are “double-colon” rules, thus

```
install :: $(HS_PROG)
      ...how to install it...
```

GNU **make** treats double-colon rules as separate entities. If there are several double-colon rules for the same target it takes each in turn and fires it if its dependencies say to do so. This means that you can, for example, define both `HS_PROG` and `LIBRARY`, which will generate two rules for `install`. When you type **gmake install** both rules will be fired, and both the program and the library will be installed, just as you wanted.

8.9. Recursion

In leaf `Makefiles` the variable `SUBDIRS` is undefined. In non-leaf `Makefiles`, `SUBDIRS` is set to the list of sub-directories that contain subordinate `Makefiles`. *It is up to you to set `SUBDIRS` in the `Makefile`.* There is no automation here—`SUBDIRS` is too important to automate.

When `SUBDIRS` is defined, `target.mk` includes a rather neat rule for the standard targets (Section 7.7, "Standard Targets" that simply invokes **make** recursively in each of the sub-directories.

These recursive invocations are guaranteed to occur in the order in which the list of directories is specified in `SUBDIRS`. This guarantee can be important. For example, when you say **gmake boot** it can be important that the recursive invocation of **make boot** is done in one sub-directory (the include files, say) before another (the source files). Generally, put the most independent sub-directory first, and the most dependent last.

8.10. Way management

We sometimes want to build essentially the same system in several different "ways". For example, we want to build GHC's `Prelude` libraries with and without profiling, so that there is an appropriately-built library archive to link with when the user compiles his program. It would be possible to have a completely separate build tree for each such "way", but it would be horribly bureaucratic, especially since often only parts of the build tree need to be constructed in multiple ways.

Instead, the `target.mk` contains some clever magic to allow you to build several versions of a system; and to control locally how many versions are built and how they differ. This section explains the magic.

The files for a particular way are distinguished by munging the suffix. The "normal way" is always built, and its files have the standard suffices `.o`, `.hi`, and so on. In addition, you can build one or more extra ways, each distinguished by a *way tag*. The object files and interface files for one of these extra ways are distinguished by their suffix. For example, way `mp` has files `.mp_o` and `.mp_hi`. Library archives have their way tag the other side of the dot, for boring reasons; thus, `libHS_mp.a`.

A **make** variable called `way` holds the current way tag. *`way` is only ever set on the command line of **gmake*** (usually in a recursive invocation of **gmake** by the system). It is never set inside a `Makefile`. So it is a global constant for any one invocation of **gmake**. Two other **make** variables, `way_` and `_way` are immediately derived from `$(way)` and never altered. If `way` is not set, then neither are `way_` and `_way`, and the invocation of **make** will build the "normal way". If `way` is set, then the other two variables are set in sympathy. For example, if `$(way)` is "`mp`", then `way_` is set to "`mp_`" and `_way` is set to "`_mp`". These three variables are then used when constructing file names.

So how does **make** ever get recursively invoked with `way` set? There are two ways in which this happens:

- For some (but not all) of the standard targets, when in a leaf sub-directory, **make** is recursively invoked for each way tag in `$(WAYS)`. You set `WAYS` in the `Makefile` to the list of way tags you want these targets built for. The mechanism here is very much like the recursive invocation of **make** in sub-directories (Section 8.9, "Recursion"). It is up to you to set `WAYS` in your `Makefile`; this is how you control what ways will get built.

- For a useful collection of targets (such as `libHS_mp.a`, `Foo.mp_o`) there is a rule which recursively invokes **make** to make the specified target, setting the `way` variable. So if you say **gmake Foo.mp_o** you should see a recursive invocation **gmake Foo.mp_o way=mp**, and *in this recursive invocation the pattern rule for compiling a Haskell file into a `.o` file will match.* The key pattern rules (in `suffix.mk`) look like this:

```
%.$(way_)o : %.lhs
      $(HC) $(HC_OPTS) $< -o $@
```

Neat, eh?

- You can invoke **make** with a particular `way` setting yourself, in order to build files related to a particular `way` in the current directory. eg.

  ```
  $ make way=p
  ```

  will build files for the profiling way only in the current directory.

8.11. When the canned rule isn't right

Sometimes the canned rule just doesn't do the right thing. For example, in the `nofib` suite we want the link step to print out timing information. The thing to do here is *not* to define `HS_PROG` or `C_PROG`, and instead define a special purpose rule in your own `Makefile`. By using different variable names you will avoid the canned rules being included, and conflicting with yours.

9. Building the documentation

9.1. Tools for building the Documentation

The following additional tools are required if you want to format the documentation that comes with the `fptools` projects:

DocBook
: Much of our documentation is written in DocBook XML, instructions on installing and configuring the DocBook tools are below.

TeX
: A decent TeX distribution is required if you want to produce printable documentation. We recomment teTeX, which includes just about everything you need.

Haddock
: Haddock is a Haskell documentation tool that we use for automatically generating documentation from the library source code. It is an `fptools` project in itself. To build documentation for the libraries (`fptools/libraries`) you should check out and build Haddock in `fptools/haddock`. Haddock requires GHC to build.

9.2. Installing the DocBook tools

9.2.1. Installing the DocBook tools on Linux

If you're on a recent RedHat (7.0+) or SuSE (8.1+) system, you probably have working DocBook tools already installed. The configure script should detect your setup and you're away.

If you don't have DocBook tools installed, and you are using a system that can handle RPM packages, you can use Rpmfind.net [http://rpmfind.net/] to find suitable packages for your system. Search for the packages `docbook-dtd`, `docbook-xsl-stylesheets`, `libxslt`, `libxml2`, `fop`, `xmltex`, and `dvips`.

9.2.2. Installing DocBook on FreeBSD

On FreeBSD systems, the easiest way to get DocBook up and running is to install it from the ports tree or a pre-compiled package (packages are available from your local FreeBSD mirror site).

To use the ports tree, do this:

```
$ cd /usr/ports/textproc/docproj
$ make install
```

This installs the FreeBSD documentation project tools, which includes everything needed to format the GHC documentation.

9.2.3. Installing from binaries on Windows

Probably the fastest route to a working DocBook environment on Windows is to install Cygwin [http://www.cygwin.com/] with the complete `Doc` category. If you are using MinGW [http://www.mingw.org/] for compilation, you have to help **configure** a little bit: Set the environment variables `XmllintCmd` and `XsltprocCmd` to the paths of the Cygwin executables **xmllint** and **xsltproc**, respectively, and set `fp_cv_dir_docbook_xsl` to the path of the directory where the XSL stylesheets are installed, e.g. `c:/cygwin/usr/share/docbook-xsl`.

If you want to build HTML Help, you have to install the HTML Help SDK [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/hworiHTMLHelpStartPage.asp], too, and make sure that **hhc** is in your `PATH`.

9.3. Configuring the DocBook tools

Once the DocBook tools are installed, the configure script will detect them and set up the build system accordingly. If you have a system that isn't supported, let us know, and we'll try to help.

9.4. Building the documentation

To build documentation in a certain format, you can say, for example,

```
$ make html
```

to build HTML documentation below the current directory. The available formats are: `dvi`, `ps`, `pdf`, `html`, and `rtf`. Note that not all documentation can be built in all of these formats: HTML documentation is generally supported everywhere, and DocBook documentation might support the other formats (depending on what other tools you have installed).

All of these targets are recursive; that is, saying `make html` will make HTML docs for all the documents recursively below the current directory.

Because there are many different formats that the DocBook documentation can be generated in, you have to select which ones you want by setting the `XMLDocWays` variable to a list of them. For example, in `build.mk` you might have a line:

```
XMLDocWays = html ps
```

This will cause the documentation to be built in the requested formats as part of the main build (the default is not to build any documentation at all).

9.5. Installing the documentation

To install the documentation, use:

```
$ make install-docs
```

This will install the documentation into `$(datadir)` (which defaults to `$(prefix)/share`). The exception is HTML documentation, which goes into `$(datadir)/html`, to keep things tidy.

Note that unless you set `$(XMLDocWays)` to a list of formats, the `install-docs` target won't do anything for DocBook XML documentation.

10. Porting GHC

This section describes how to port GHC to a currenly unsupported platform. There are two distinct possibilities:

- The hardware architecture for your system is already supported by GHC, but you're running an OS that isn't supported (or perhaps has been supported in the past, but currently isn't). This is the easiest type of porting job, but it still requires some careful bootstrapping. Proceed to Section 10.1, "Booting/porting from C (`.hc`) files".
- Your system's hardware architecture isn't supported by GHC. This will be a more difficult port (though by comparison perhaps not as difficult as porting gcc). Proceed to Section 10.2, "Porting GHC to a new architecture".

10.1. Booting/porting from C (`.hc`) files

Bootstrapping GHC on a system without GHC already installed is achieved by taking the intermediate C files (known as HC files) from another GHC compilation, compiling them using gcc to get a working GHC.

NOTE: GHC versions 5.xx were hard to bootstrap from C. We recommend using GHC 6.0.1 or later.

HC files are platform-dependent, so you have to get a set that were generated on *the same platform*. There may be some supplied on the GHC download page, otherwise you'll have to compile some up yourself, or start from *unregisterised* HC files - see Section 10.2, "Porting GHC to a new architecture".

The following steps should result in a working GHC build with full libraries:

- Unpack the HC files on top of a fresh source tree (make sure the source tree version matches the version of the HC files *exactly*!). This will place matching `.hc` files next to the corresponding Haskell source (`.hs` or `.lhs`) in the compiler subdirectory `ghc/compiler` and in the libraries (subdirectories of `libraries`).
- The actual build process is fully automated by the `hc-build` script located in the `distrib` directory. If you eventually want to install GHC into the directory *`dir`*, the following command will execute the whole build process (it won't install yet):

  ```
  $ distrib/hc-build --prefix=dir
  ```

  By default, the installation directory is `/usr/local`. If that is what you want, you may omit the argument to `hc-build`. Generally, any option given to `hc-build` is passed through to the configuration script `configure`. If `hc-build` successfully completes the build process, you can install the resulting system, as normal, with

```
$ make install
```

10.2. Porting GHC to a new architecture

The first step in porting to a new architecture is to get an *unregisterised* build working. An unregisterised build is one that compiles via vanilla C only. By contrast, a registerised build uses the following architecture-specific hacks for speed:

- Global register variables: certain abstract machine "registers" are mapped to real machine registers, depending on how many machine registers are available (see `ghc/includes/MachRegs.h`).
- Assembly-mangling: when compiling via C, we feed the assembly generated by gcc though a Perl script known as the *mangler* (see `ghc/driver/mangler/ghc-asm.lprl`). The mangler rearranges the assembly to support tail-calls and various other optimisations.

In an unregisterised build, neither of these hacks are used — the idea is that the C code generated by the compiler should compile using gcc only. The lack of these optimisations costs about a factor of two in performance, but since unregisterised compilation is usually just a step on the way to a full registerised port, we don't mind too much.

Notes on GHC portability in general: we've tried to stick to writing portable code in most parts of the system, so it should compile on any POSIXish system with gcc, but in our experience most systems differ from the standards in one way or another. Deal with any problems as they arise - if you get stuck, ask the experts on `<glasgow-haskell-users@haskell.org>`.

Lots of useful information about the innards of GHC is available in the GHC Commentary [http://www.cse.unsw.edu.au/~chak/haskell/ghc/comm/], which might be helpful if you run into some code which needs tweaking for your system.

10.2.1. Cross-compiling to produce an unregisterised GHC

NOTE! These instructions apply to GHC 6.4 and (hopefully) later. If you need instructions for an earlier version of GHC, try to get hold of the version of this document that was current at the time. It should be available from the appropriate download page on the GHC homepage [http://www.haskell.org/ghc/].

In this section, we explain how to bootstrap GHC on a new platform, using unregisterised intermediate C files. We haven't put a great deal of effort into automating this process, for two reasons: it is done very rarely, and the process usually requires human intervention to cope with minor porting issues anyway.

The following step-by-step instructions should result in a fully working, albeit unregisterised, GHC. Firstly, you need a machine that already has a working GHC (we'll call this the *host* machine), in order to cross-compile the intermediate C files that we will use to bootstrap the compiler on the *target* machine.

- On the target machine:
    - Unpack a source tree (preferably a released version). We will call the path to the root of this tree *`T`*.
    -

```
$ cd T
$ ./configure --enable-hc-boot --enable-hc-boot-unregisterised
```

You might need to update `configure.in` to recognise the new architecture, and re-generate `configure` with `autoreconf`.

- ```
  $ cd T/ghc/includes
  $ make
  ```

- On the host machine:

  - Unpack a source tree (same released version). Call this directory *H*.

  - ```
    $ cd H
    $ ./configure
    ```

  - Create *H*/`mk/build.mk`, with the following contents:

    ```
    GhcUnregisterised = YES
    GhcLibHcOpts = -O -fvia-C -keep-hc-files
    GhcRtsHcOpts = -keep-hc-files
    GhcLibWays =
    SplitObjs = NO
    GhcWithNativeCodeGen = NO
    GhcWithInterpreter = NO
    GhcStage1HcOpts = -O
    GhcStage2HcOpts = -O -fvia-C -keep-hc-files
    SRC_HC_OPTS += -H32m
    GhcBootLibs = YES
    ```

  - Edit *H*/`mk/config.mk`:

    - change `TARGETPLATFORM` appropriately, and set the variables involving `TARGET` to the correct values for the target platform. This step is necessary because currently `configure` doesn't cope with specifying different values for the `--host` and `--target` flags.

    - copy `LeadingUnderscore` setting from target.

  - Copy *T*/`ghc/includes/ghcautoconf.h`, *T*/`ghc/includes/DerivedConstants.h`, and *T*/`ghc/includes/GHCConstants.h` to *H*/`ghc/includes`. Note that we are building on the host machine, using the target machine's configuration files. This is so that the intermediate C files generated here will be suitable for compiling on the target system.

  - Touch the generated configuration files, just to make sure they don't get replaced during the build:

    ```
    $ cd H/ghc/includes
    $ touch ghcautoconf.h DerivedConstants.h GHCConstants.h mkDerivedConstants.c
    $ touch mkDerivedConstantsHdr mkDerivedConstants.o mkGHCConstants mkGHCConstants.o
    ```

    Note: it has been reported that these files still get overwritten during the next stage. We have installed a fix for this in GHC 6.4.2, but if you are building a version before that you need to watch out for these files getting overwritte by the `Makefile` in `ghc/includes`. If your system supports it, you might be able to prevent it by making them immutable:

```
$ chflags uchg  ghc/includes/{ghcautoconf.h,DerivedConstants.h,GHCConstants.h}
```

- Now build the compiler:

  ```
  $ cd H/glafp-utils && make boot && make
  $ cd H/ghc && make boot && make
  ```

  Don't worry if the build falls over in the RTS, we don't need the RTS yet.

- ```
  $ cd H/libraries
  $ make boot && make
  ```

- ```
  $ cd H/ghc/compiler
  $ make boot stage=2 && make stage=2
  ```

- ```
  $ cd H/ghc/lib/compat
  $ make clean
  $ rm .depend
  $ make boot UseStage1=YES
  $ make -k UseStage1=YES EXTRA_HC_OPTS='-O -fvia-C -keep-hc-files'
  $ cd H/ghc/utils
  $ make clean
  $ make -k UseStage1=YES EXTRA_HC_OPTS='-O -fvia-C -keep-hc-files'
  ```

- ```
  $ cd H
  $ make hc-file-bundle Project=Ghc
  ```

- copy *H*`/*-hc.tar.gz` to *T*`/...`

- On the target machine:

  At this stage we simply need to bootstrap a compiler from the intermediate C files we generated above. The process of bootstrapping from C files is automated by the script in `distrib/hc-build`, and is described in Section 10.1, "Booting/porting from C (`.hc`) files".

  ```
  $ ./distrib/hc-build --enable-hc-boot-unregisterised
  ```

  However, since this is a bootstrap on a new machine, the automated process might not run to completion the first time. For that reason, you might want to treat the `hc-build` script as a list of instructions to follow, rather than as a fully automated script. This way you'll be able to restart the process part-way through if you need to fix anything on the way.

  Don't bother with running `make install` in the newly bootstrapped tree; just use the compiler in that tree to build a fresh compiler from scratch, this time without booting from C files. Before doing this, you might want to check that the bootstrapped compiler is generating working binaries:

```
$ cat >hello.hs
main = putStrLn "Hello World!\n"
^D
$ T/ghc/compiler/ghc-inplace hello.hs -o hello
$ ./hello
Hello World!
```

Once you have the unregisterised compiler up and running, you can use it to start a registerised port. The following sections describe the various parts of the system that will need architecture-specific tweaks in order to get a registerised build going.

10.2.2. Porting the RTS

The following files need architecture-specific code for a registerised build:

`ghc/includes/MachRegs.h`
: Defines the STG-register to machine-register mapping. You need to know your platform's C calling convention, and which registers are generally available for mapping to global register variables. There are plenty of useful comments in this file.

`ghc/includes/TailCalls.h`
: Macros that cooperate with the mangler (see Section 10.2.3, “The mangler”) to make proper tail-calls work.

`ghc/rts/Adjustor.c`
: Support for `foreign import "wrapper"` (aka `foreign export dynamic`). Not essential for getting GHC bootstrapped, so this file can be deferred until later if necessary.

`ghc/rts/StgCRun.c`
: The little assembly layer between the C world and the Haskell world. See the comments and code for the other architectures in this file for pointers.

`ghc/rts/MBlock.h`, `ghc/rts/MBlock.c`
: These files are really OS-specific rather than architecture-specific. In `MBlock.h` is specified the absolute location at which the RTS should try to allocate memory on your platform (try to find an area which doesn't conflict with code or dynamic libraries). In `Mblock.c` you might need to tweak the call to `mmap()` for your OS.

10.2.3. The mangler

The mangler is an evil Perl-script (`ghc/driver/mangler/ghc-asm.lprl`) that rearranges the assembly code output from gcc to do two main things:

- Remove function prologues and epilogues, and all movement of the C stack pointer. This is to support tail-calls: every code block in Haskell code ends in an explicit jump, so we don't want the C-stack overflowing while we're jumping around between code blocks.
- Move the *info table* for a closure next to the entry code for that closure. In unregisterised code, info tables contain a pointer to the entry code, but in registerised compilation we arrange that the info table is shoved right up against the entry code, and addressed backwards from the entry code pointer (this saves a word in the info table and an extra indirection when jumping to the closure entry code).

The mangler is abstracted to a certain extent over some architecture-specific things such as the particular assembler directives used to herald symbols. Take a look at the definitions for other architectures and use these as a starting point.

10.2.4. The splitter

The splitter is another evil Perl script (`ghc/driver/split/ghc-split.lprl`). It cooperates with the mangler to support object splitting. Object splitting is what happens when the `-split-objs` option is passed to GHC: the object file is split into many smaller objects. This feature is used when building libraries, so that a program statically linked against the library will pull in less of the library.

The splitter has some platform-specific stuff; take a look and tweak it for your system.

10.2.5. The native code generator

The native code generator isn't essential to getting a registerised build going, but it's a desirable thing to have because it can cut compilation times in half. The native code generator is described in some detail in the GHC commentary [http://www.cse.unsw.edu.au/~chak/haskell/ghc/comm/].

10.2.6. GHCi

To support GHCi, you need to port the dynamic linker (`fptools/ghc/rts/Linker.c`). The linker currently supports the ELF and PEi386 object file formats - if your platform uses one of these then things will be significantly easier. The majority of Unix platforms use the ELF format these days. Even so, there are some machine-specific parts of the ELF linker: for example, the code for resolving particular relocation types is machine-specific, so some porting of this code to your architecture will probaly be necessary.

If your system uses a different object file format, then you have to write a linker — good luck!

11. Known pitfalls in building Glasgow Haskell

WARNINGS about pitfalls and known “problems”:

1. One difficulty that comes up from time to time is running out of space in `TMPDIR`. (It is impossible for the configuration stuff to compensate for the vagaries of different sysadmin approaches to temp space.) The quickest way around it is **setenv TMPDIR /usr/tmp** or even **setenv TMPDIR .** (or the equivalent incantation with your shell of choice). The best way around it is to say

   ```
   export TMPDIR=<dir>
   ```

   in your `build.mk` file. Then GHC and the other `fptools` programs will use the appropriate directory in all cases.

2. In compiling some support-code bits, e.g., in `ghc/rts/gmp` and even in `ghc/lib`, you may get a few C-compiler warnings. We think these are OK.

3. When compiling via C, you'll sometimes get “warning: assignment from incompatible pointer type” out of GCC. Harmless.

4. Similarly, **ar**chiving warning messages like the following are not a problem:

   ```
   ar: filename GlaIOMonad__1_2s.o truncated to GlaIOMonad_
   ar: filename GlaIOMonad__2_2s.o truncated to GlaIOMonad_
   ...
   ```

5. In compiling the compiler proper (in compiler/), you *may* get an "Out of heap space" error message. These can vary with the vagaries of different systems, it seems. The solution is simple:

   - If you're compiling with GHC 4.00 or later, then the *maximum* heap size must have been reached. This is somewhat unlikely, since the maximum is set to 64M by default. Anyway, you can raise it with the -optCrts-M<size> flag (add this flag to <module>_HC_OPTS **make** variable in the appropriate Makefile).

   - For GHC < 4.00, add a suitable -H flag to the Makefile, as above.

   and try again: **gmake**. (see Section 8.7, "Pattern rules and options" for information about <module>_HC_OPTS.) Alternatively, just cut to the chase:

   ```
   $ cd ghc/compiler
   $ make EXTRA_HC_OPTS=-optCrts-M128M
   ```

6. If you try to compile some Haskell, and you get errors from GCC about lots of things from /usr/include/math.h, then your GCC was mis-installed. **fixincludes** wasn't run when it should've been. As **fixincludes** is now automagically run as part of GCC installation, this bug also suggests that you have an old GCC.

7. You *may* need to re-**ranlib** your libraries (on Sun4s).

   ```
   $ cd $(libdir)/ghc-x.xx/sparc-sun-sunos4
   $ foreach i ( `find . -name '*.a' -print` ) # or other-shell equiv...
   ?    ranlib $i
   ?    # or, on some machines: ar s $i
   ? end
   ```

   We'd be interested to know if this is still necessary.

8. GHC's sources go through **cpp** before being compiled, and **cpp** varies a bit from one Unix to another. One particular gotcha is macro calls like this:

   ```
   SLIT("Hello, world")
   ```

   Some **cpp**s treat the comma inside the string as separating two macro arguments, so you get

   ```
   :731: macro `SLIT' used with too many (2) args
   ```

   Alas, **cpp** doesn't tell you the offending file! Workaround: don't put weird things in string args to **cpp** macros.

12. Platforms, scripts, and file names

GHC is designed both to be built, and to run, on both Unix and Windows. This flexibility gives rise to a good deal of brain-bending detail, which we have tried to collect in this chapter.

12.1. Windows platforms: Cygwin, MSYS, and MinGW

The build system is built around Unix-y makefiles. Because it's not native, the Windows situation for building GHC is particularly confusing. This section tries to clarify, and to establish terminology.

12.1.1. MinGW

MinGW (Minimalist GNU for Windows) [http://www.mingw.org] is a collection of header files and import libraries that allow one to use **gcc** and produce native Win32 programs that do not rely on any third-party DLLs. The current set of tools include GNU Compiler Collection (**gcc**), GNU Binary Utilities (Binutils), GNU debugger (Gdb), GNU make, and a assorted other utilities.

The down-side of MinGW is that the MinGW libraries do not support anything like the full Posix interface.

12.1.2. Cygwin and MSYS

You can't use the MinGW to *build* GHC, because MinGW doesn't have a shell, or the standard Unix commands such as **mv**, **rm**, **ls**, nor build-system stuff such as **make** and **darcs**. For that, there are two choices: Cygwin [http://www.cygwin.com] and MSYS [http://www.mingw.org/msys.shtml]:

- Cygwin comes with compilation tools (**gcc**, **ld** and so on), which compile code that has access to all of Posix. The price is that the executables must be dynamically linked with the Cygwin DLL, so that *you cannot run a Cywin-compiled program on a machine that doesn't have Cygwin*. Worse, Cygwin is a moving target. The name of the main DLL, `cygwin1.dll` does not change, but the implementation certainly does. Even the interfaces to functions it exports seem to change occasionally.
- MSYS is a fork of the Cygwin tree, so they are fundamentally similar. However, MSYS is by design much smaller and simpler. Access to the file system goes through fewer layers, so MSYS is quite a bit faster too.

  Furthermore, MSYS provides no compilation tools; it relies instead on the MinGW tools. These compile binaries that run with no DLL support, on any Win32 system. However, MSYS does come with all the make-system tools, such as **make**, **autoconf**, **darcs**, **ssh** etc. To get these, you have to download the MsysDTK (Developer Tool Kit) package, as well as the base MSYS package.

  MSYS does have a DLL, but it's only used by MSYS commands (**sh**, **rm**, **ssh** and so on), not by programs compiled under MSYS.

12.1.3. Targeting MinGW

We want GHC to compile programs that work on any Win32 system. Hence:

- GHC does invoke a C compiler, assembler, linker and so on, but we ensure that it only invokes the MinGW tools, not the Cygwin ones. That means that the programs GHC compiles will work on any system, but it also means that the programs GHC compiles do not have access to all of Posix. In particular, they cannot import the (Haskell) Posix library; they have to do their input output using standard Haskell I/O libraries, or native Win32 bindings.

  We will call a GHC that targets MinGW in this way *GHC-mingw*.
- To make the GHC distribution self-contained, the GHC distribution includes the MinGW **gcc**, **as**, **ld**, and a bunch of input/output libraries.

So *GHC targets MinGW*, not Cygwin. It is in principle possible to build a version of GHC, *GHC-cygwin*, that targets Cygwin instead. The up-side of GHC-cygwin is that Haskell programs compiled by GHC-cygwin can import the (Haskell) Posix library. *We do not support GHC-cygwin, however; it is beyond our resources.*

While GHC *targets* MinGW, that says nothing about how GHC is *built*. We use both MSYS and Cygwin as build environments for GHC; both work fine, though MSYS is rather lighter weight.

In your build tree, you build a compiler called **ghc-inplace**. It uses the **gcc** that you specify using the `--with-gcc` flag when you run **configure** (see below). The makefiles are careful to use **ghc-inplace** (not **gcc**) to compile any C files, so that it will in turn invoke the correct **gcc** rather that whatever one happens to be in your path. However, the makefiles do use whatever **ld** and **ar** happen to be in your path. This is a bit naughty, but (a) they are only used to glom together .o files into a bigger .o file, or a .a file, so they don't ever get libraries (which would be bogus; they might be the wrong libraries), and (b) Cygwin and MinGW use the same .o file format. So its ok.

12.1.4. File names

Cygwin, MSYS, and the underlying Windows file system all understand file paths of form `c:/tmp/foo`. However:

- MSYS programs understand `/bin`, `/usr/bin`, and map Windows's lettered drives as `/c/tmp/foo` etc. The exact mount table is given in the doc subdirectory of the MSYS distribution.

  When it invokes a command, the MSYS shell sees whether the invoked binary lives in the MSYS `/bin` directory. If so, it just invokes it. If not, it assumes the program is no an MSYS program, and walks over the command-line arguments changing MSYS paths into native-compatible paths. It does this inside sub-arguments and inside quotes. For example, if you invoke

  ```
  foogle -B/c/tmp/baz
  ```

  the MSYS shell will actually call `foogle` with argument `-Bc:/tmp/baz`.

- Cygwin programs have a more complicated mount table, and map the lettered drives as `/cygdrive/c/tmp/foo`.

  The Cygwin shell does no argument processing when invoking non-Cygwin programs.

12.1.5. Crippled ld

It turns out that on both Cygwin and MSYS, the **ld** has a limit of 32kbytes on its command line. Especially when using split object files, the make system can emit calls to **ld** with thousands of files on it. Then you may see something like this:

```
(cd Graphics/Rendering/OpenGL/GL/QueryUtils_split && /mingw/bin/ld -r -x -o ../QueryUtils.o *.o)
/bin/sh: /mingw/bin/ld: Invalid argument
```

The solution is either to switch off object file splitting (set `SplitObjs` to `NO` in your `build.mk`), or to make the module smaller.

12.1.6. Host System vs Target System

In the source code you'll find various ifdefs looking like:

```
#ifdef mingw32_HOST_OS
  ...blah blah...
#endif
```

and

```
#ifdef mingw32_TARGET_OS
  ...blah blah...
#endif
```

These macros are set by the configure script (via the file config.h). Which is which? The criterion is this. In the ifdefs in GHC's source code:

- The "host" system is the one on which GHC itself will be run.
- The "target" system is the one for which the program compiled by GHC will be run.

For a stage-2 compiler, in which GHCi is available, the "host" and "target" systems must be the same. So then it doesn't really matter whether you use the HOST_OS or TARGET_OS cpp macros.

12.2. Wrapper scripts

Many programs, including GHC itself and hsc2hs, need to find associated binaries and libraries. For *installed* programs, the strategy depends on the platform. We'll use GHC itself as an example:

- On Unix, the command **ghc** is a shell script, generated by adding installation paths to the front of the source file `ghc.sh`, that invokes the real binary, passing "-B*path*" as an argument to tell **ghc** where to find its supporting files.
- On vanilla Windows, it turns out to be much harder to make reliable script to be run by the native Windows shell **cmd** (e.g. limits on the length of the command line). So instead we invoke the GHC binary directly, with no -B flag. GHC uses the Windows `getExecDir` function to find where the executable is, and from that figures out where the supporting files are.

(You can find the layout of GHC's supporting files in the section "Layout of installed files" of Section 2 of the GHC user guide.)

Things work differently for *in-place* execution, where you want to execute a program that has just been built in a build tree. The difference is that the layout of the supporting files is different. In this case, whether on Windows or Unix, we always use a shell script. This works OK on Windows because the script is executed by MSYS or Cygwin, which don't have the shortcomings of the native Windows **cmd** shell.

13. Instructions for building under Windows

This section gives detailed instructions for how to build GHC from source on your Windows machine. Similar instructions for installing and running GHC may be found in the user guide. In general, Win95/Win98 behave the same, and WinNT/Win2k behave the same.

Make sure you read the preceding section on platforms (Section 12, “Platforms, scripts, and file names”) before reading section. You don't need Cygwin or MSYS to *use* GHC, but you do need one or the other to *build* GHC.

13.1. Installing and configuring MSYS

MSYS is a lightweight alternative to Cygwin. You don't need MSYS to *use* GHC, but you do need it or Cygwin to *build* GHC. Here's how to install MSYS.

- Go to http://www.mingw.org/download.shtml and download the following (of course, the version numbers will differ):
  - The main MSYS package (binary is sufficient): `MSYS-1.0.9.exe`
  - The MSYS developer's toolkit (binary is sufficient): `msysDTK-1.0.1.exe`. This provides **make**, **autoconf**, **ssh** and probably more besides.

  Run both executables (in the order given above) to install them. I put them in `c:/msys`

- Set the following environment variables

  - `PATH`: add `c:/msys/1.0/bin` and `c:/msys/1.0/local/bin` to your path. (Of course, the version number may differ.) MSYS mounts the former as both `/bin` and `/usr/bin` and the latter as `/usr/local/bin`.

  - `HOME`: set to your home directory (e.g. `c:/userid`). This is where, among other things, **ssh** will look for your `.ssh` directory.

  - `SHELL`: set to `c:/msys/1.0/bin/sh.exe`

  - `CVS_RSH`: set to `c:/msys/1.0/bin/ssh.exe`. Only necessary if you are using CVS.

  - `MAKE_MODE`: set to `UNIX`. (I'm not certain this is necessary for MSYS.)

- Check that the `CYGWIN` environment variable is *not* set. It's a bad bug that MSYS is affected by this, but if you have CYGWIN set to "ntsec ntea", which is right for Cygwin, it causes the MSYS **ssh** to bogusly fail complaining that your `.ssh/identity` file has too-liberal permissinos.

Here are some points to bear in mind when using MSYS:

- MSYS does some kind of special magic to binaries stored in `/bin` and `/usr/bin`, which are by default both mapped to `c:/msys/1.0/bin` (assuming you installed MSYS in `c:/msys`). Do not put any other binaries (such as GHC or Alex) in this directory or its sub-directories: they fail in mysterious ways. However, it's fine to put other binaries in `/usr/local/bin`, which maps to `c:/msys/1.0/local/bin`.

- MSYS seems to implement symbolic links by copying, so sharing is lost.

- Win32 has a **find** command which is not the same as MSYS's find. You will probably discover that the Win32 **find** appears in your `PATH` before the MSYS one, because it's in the *system* `PATH` environment variable, whereas you have probably modified the *user* `PATH` variable. You can always invoke **find** with an absolute path, or rename it.

- MSYS comes with **bzip**, and MSYS's **tar**'s `-j` will bunzip an archive (e.g. `tar xvjf foo.tar.bz2`). Useful when you get a bzip'd dump.

13.2. Installing and configuring Cygwin

Install Cygwin from http://www.cygwin.com/. The installation process is straightforward; we install it in `c:/cygwin`.

You must install enough Cygwin *packages* to support building GHC. If you miss out any of these, strange things will happen to you. There are two ways to do this:

- The direct, but laborious way is to select all of the following packages in the installation dialogue: **cvs**, **openssh**, **autoconf**, **binutils** (includes ld and (I think) ar), **gcc**, **flex**, **make**. To see thse packages, click on the "View" button in the "Select Packages" stage of Cygwin's installation dialogue, until the view says "Full". The default view, which is "Category" isn't very helpful, and the "View" button is rather unobtrousive.

- The clever way is to point the Cygwin installer at the **ghc-depends** package, which is kept at http://haskell.org/ghc/cygwin. When the Cygwin installer asks you to "Choose a Download Site", choose one of the offered mirror sites; and then type "http://haskell.org/ghc/cygwin" into the "User

URL" box and click "Add"; now two sites are selected. (The Cygwin installer remembers this for next time.) Click "Next".

In the "Select Packages" dialogue box that follows, click the "+" sign by "Devel", scroll down to the end of the "Devel" packages, and choose **ghc-depends**. The package **ghc-depends** will not actually install anything itself, but forces additional packages to be added by the Cygwin installer.

Now set the following user environment variables:

- Add `c:/cygwin/bin` and `c:/cygwin/usr/bin` to your `PATH`
- Set `MAKE_MODE` to `UNIX`. If you don't do this you get very weird messages when you type **make**, such as:

  ```
  /c: /c: No such file or directory
  ```

- Set `SHELL` to `c:/cygwin/bin/bash`. When you invoke a shell in Emacs, this `SHELL` is what you get.
- Set `HOME` to point to your home directory. This is where, for example, **bash** will look for your `.bashrc` file. Ditto **emacs** looking for `.emacsrc`

Here are some things to be aware of when using Cygwin:

- Cygwin doesn't deal well with filenames that include spaces. "`Program Files`" and "`Local files`" are common gotchas.
- Cygwin implements a symbolic link as a text file with some magical text in it. So other programs that don't use Cygwin's I/O libraries won't recognise such files as symlinks. In particular, programs compiled by GHC are meant to be runnable without having Cygwin, so they don't use the Cygwin library, so they don't recognise symlinks.
- See the notes in Section 13.1, "Installing and configuring MSYS" about **find** and **bzip**, which apply to Cygwin too.
- Some script files used in the make system start with "**#!/bin/perl**", (and similarly for **sh**). Notice the hardwired path! So you need to ensure that your `/bin` directory has at least **sh**, **perl**, and **cat** in it. All these come in Cygwin's `bin` directory, which you probably have installed as `c:/cygwin/bin`. By default Cygwin mounts "`/`" as `c:/cygwin`, so if you just take the defaults it'll all work ok. (You can discover where your Cygwin root directory `/` is by typing **mount**.) Provided `/bin` points to the Cygwin `bin` directory, there's no need to copy anything. If not, copy these binaries from the `cygwin/bin` directory (after fixing the `sh.exe` stuff mentioned in the previous bullet).
- By default, cygwin provides the command shell `ash` as `sh.exe`. It seems to be fine now, but in the past we saw build-system problems that turned out to be due to bugs in `ash` (to do with quoting and length of command lines). On the other hand `bash` seems to be rock solid. If this happens to you (which it shouldn't), in `cygwin/bin` remove the supplied `sh.exe` (or rename it as `ash.exe`), and copy `bash.exe` to `sh.exe`. You'll need to do this in Windows Explorer or the Windows **cmd** shell, because you can't rename a running program!

13.3. Configuring SSH

ssh comes with both Cygwin and MSYS. (Cygwin note: you need to ask for package **openssh** (not ssh) in the Cygwin list of packages; or use the **ghc-depends** package -- see Section 13.2, “Installing and configuring Cygwin”.)

There are several strange things about **ssh** on Windows that you need to know.

- The programs **ssh-keygen1**, **ssh1**, and **cvs**, seem to lock up **bash** entirely if they try to get user input (e.g. if they ask for a password). To solve this, start up `cmd.exe` and run it as follows:

  ```
  c:\tmp> set CYGWIN32=tty
  c:\tmp> c:/user/local/bin/ssh-keygen1
  ```

- (Cygwin-only problem, I think.) **ssh** needs to access your directory `.ssh`, in your home directory. To determine your home directory **ssh** first looks in `c:/cygwin/etc/passwd` (or wherever you have Cygwin installed). If there's an entry there with your userid, it'll use that entry to determine your home directory, *ignoring the setting of the environment variable $HOME*. If the home directory is bogus, **ssh** fails horribly. The best way to see what is going on is to say

  ```
  ssh -v cvs.haskell.org
  ```

  which makes **ssh** print out information about its activity.

  You can fix this problem, either by correcting the home-directory field in `c:/cygwin/etc/passwd`, or by simply deleting the entire entry for your userid. If you do that, **ssh** uses the $HOME environment variable instead.

- To protect your `.ssh` from access by anyone else, right-click your `.ssh` directory, and select `Properties`. If you are not on the access control list, add yourself, and give yourself full permissions (the second panel). Remove everyone else from the access control list. Don't leave them there but deny them access, because 'they' may be a list that includes you!

- In fact **ssh** 3.6.1 now seems to *require* you to have Unix permissions 600 (read/write for owner only) on the `.ssh/identity` file, else it bombs out. For your local C drive, it seems that `chmod 600 identity` works, but on Windows NT/XP, it doesn't work on a network drive (exact dteails obscure). The solution seems to be to set the $CYGWIN environment variable to "`ntsec neta`". The $CYGWIN environment variable is discussed in the Cygwin User's Guide [http://cygwin.com/cygwin-ug-net/using-cygwinenv.html], and there are more details in the Cygwin FAQ [http://cygwin.com/faq/faq_4.html#SEC44].

13.4. Other things you need to install

You have to install the following other things to build GHC, listed below.

On Windows you often install executables in directories with spaces, such as "`Program Files`". However, the `make` system for fptools doesn't deal with this situation (it'd have to do more quoting of binaries), so you are strongly advised to put binaries for all tools in places with no spaces in their path. On both MSYS and Cygwin, it's perfectly OK to install such programs in the standard Unixy places, `/usr/local/bin` and `/usr/local/lib`. But it doesn't matter, provided they are in your path.

- Install an executable GHC, from http://www.haskell.org/ghc. This is what you will use to compile GHC. Add it in your `PATH`: the installer tells you the path element you need to add upon completion.

- Install an executable Happy, from http://www.haskell.org/happy. Happy is a parser generator used to compile the Haskell grammar. Under MSYS or Cygwin you can easily build it from the source distribution using

```
$ ./configure
$ make
$ make install
```

This should install it in `/usr/local/bin` (which maps to `c:/msys/1.0/local/bin` on MSYS). Make sure the installation directory is in your `PATH`.

- Install an executable Alex. This can be done by building from the source distribution in the same way as Happy. Sources are available from http://www.haskell.org/alex.

- GHC uses the *mingw* C compiler to generate code, so you have to install that (see Section 12.1, “Windows platforms: Cygwin, MSYS, and MinGW”). Just pick up a mingw bundle at http://www.mingw.org/. We install it in `c:/mingw`.

  On MSYS, add `c:/mingw/bin` to your PATH. MSYS does not provide **gcc**, **ld**, **ar**, and so on, because it just uses the MinGW ones. So you need them in your path.

  On Cygwin, do not add any of the *mingw* binaries to your path. They are only going to get used by explicit access (via the --with-gcc flag you give to **configure** later). If you do add them to your path you are likely to get into a mess because their names overlap with Cygwin binaries. On the other hand, you *do* need **ld**, **ar** (and perhaps one or two other things) in your path. The Cygwin ones are fine, but you must have them; hence needing the Cygwin binutils package.

- We use **emacs** a lot, so we install that too. When you are in `fptools/ghc/compiler`, you can use "`make tags`" to make a TAGS file for emacs. That uses the utility `fptools/ghc/utils/hasktags/hasktags`, so you need to make that first. The most convenient way to do this is by going `make boot` in `fptools/ghc`. The `make tags` command also uses **etags**, which comes with **emacs**, so you will need to add `emacs/bin` to your `PATH`.

- You might want to install GLUT in your MSYS/Cygwin installation, otherwise the GLUT package will not be built with GHC.

- Finally, check out a copy of GHC sources from the darcs repository, following the instructions at http://hackage.haskell.org/trac/ghc/wiki/GhcDarcs.

13.5. Building GHC

OK! Now go read the documentation above on building from source (Section 5, “Building from source”); the bullets below only tell you about Windows-specific wrinkles.

- If you used **autoconf** instead of **autoreconf**, you'll get an error when you run `./configure`:

```
...lots of stuff...
creating mk/config.h
mk/config.h is unchanged
configuring in ghc
running /bin/sh ./configure  --cache-file=.././config.cache --srcdir=.
./configure: ./configure: No such file or directory
configure: error: ./configure failed for ghc
```

- **autoreconf** seems to create the file `configure` read-only. So if you need to run autoreconf again (which I sometimes do for safety's sake), you get

```
/usr/bin/autoconf: cannot create configure: permission denied
```

Solution: delete `configure` first.

- After **autoreconf** run **./configure** in `fptools/` thus:

  ```
  $ ./configure --host=i386-unknown-mingw32 --with-gcc=c:/mingw/bin/gcc
  ```

  This is the point at which you specify that you are building GHC-mingw (see Section 12.1.1, “MinGW”).

  Both these options are important! It's possible to get into trouble using the wrong C compiler!

  Furthermore, it's *very important* that you specify a full MinGW path for **gcc**, not a Cygwin path, because GHC (which uses this path to invoke **gcc**) is a MinGW program and won't understand a Cygwin path. For example, if you say `--with-gcc=/mingw/bin/gcc`, it'll be interpreted as `/cygdrive/c/mingw/bin/gcc`, and GHC will fail the first time it tries to invoke it. Worse, the failure comes with no error message whatsoever. GHC simply fails silently when first invoked, typically leaving you with this:

  ```
  make[4]: Leaving directory '/cygdrive/e/fptools-stage1/ghc/rts/gmp'
  ../../ghc/compiler/ghc-inplace -optc-mno-cygwin -optc-O
    -optc-Wall -optc-W -optc-Wstrict-prototypes -optc-Wmissing-prototypes
    -optc-Wmissing-declarations -optc-Winline -optc-Waggregate-return
    -optc-Wbad-function-cast -optc-Wcast-align -optc-I../includes
    -optc-I. -optc-Iparallel -optc-DCOMPILING_RTS
    -optc-fomit-frame-pointer -O2 -static
    -package-name rts -O -dcore-lint -c Adjustor.c -o Adjustor.o
  make[2]: *** [Adjustor.o] Error 1
  make[1]: *** [all] Error 1
  make[1]: Leaving directory '/cygdrive/e/fptools-stage1/ghc'
  make: *** [all] Error 1
  ```

  Be warned!

  If you want to build GHC-cygwin (Section 12.1.2, “Cygwin and MSYS”) you'll have to do something more like:

  ```
  $ ./configure --with-gcc=...the Cygwin gcc...
  ```

- If you are paranoid, delete `config.cache` if it exists. This file occasionally remembers out-of-date configuration information, which can be really confusing.

- You almost certainly want to set

  ```
  SplitObjs = NO
  ```

  in your `build.mk` configuration file (see Section 7.3, “Getting the build you want”). This tells the build system not to split each library into a myriad of little object files, one for each function. Doing so reduces binary sizes for statically-linked binaries, but on Windows it dramatically increases the time taken to build the libraries in the first place.

- Do not attempt to build the documentation. It needs all kinds of wierd Jade stuff that we haven't worked out for Win32.

13.6. A Windows build log using Cygwin

Here is a complete, from-scratch, log of all you need to build GHC using Cygwin, kindly provided by Claus Reinke. It does not discuss alternative choices, but it gives a single path that works.

```
- Install some editor (vim, emacs, whatever)

- Install cygwin (http://www.cygwin.com)
    ; i used 1.5.16-1, installed in c:\cygwin
  - run 'setup.exe'
    Choose a Download Source:
        select 'download from internet';
    Select Root Install Directory:
        root dir: c:\cygwin;
        install for: all users;
        default file type: unix
    Select Local Package Directory
        choose a spare temporary home
    Select Your Internet Connection
        Use IE5 settings
    Choose a Download Site
        Choose your preferred main mirror and
        Add 'http://www.haskell.org/ghc/cygwin'
    Select Packages
        In addition to 'Base' (default install),
        select 'Devel->ghc-depends'

- Install mingw (http://www.mingw.org/)
    ; i used MinGW-3.1.0-1.exe
    ; installed in c:\mingw
  - you probably want to add GLUT
    ; (http://www.xmission.com/~nate/glut.html)
    ; i used glut-3.7.3-mingw32.tar

- Get recent binary snapshot of ghc-6.4.1 for mingw
    ; (http://www.haskell.org/ghc/dist/stable/dist/)
  - unpack in c:/ghc
  - add C:\ghc\ghc-6.4.1\bin to %PATH%
    (Start->Control Panel->System->Advanced->Environment Variables)

- Get darcs version of ghc
    ; also, subscribe to cvs-all@haskell.org, or follow the mailing list
    ; archive, in case you checkout a version with problems
    ; http://www.haskell.org//pipermail/cvs-all/
  - mkdir c:/fptools; cd c:/fptools
    ; (or whereever you want your darcs tree to be)
  - darcs get http://darcs.haskell.org/ghc
  - cd ghc
  - chmod +x darcs-all
  - ./darcs-all get

- Build ghc, using cygwin and mingw, targetting mingw
  - export PATH=/cygdrive/c/ghc/ghc-6.4.1:$PATH
    ; for haddock, alex, happy (*)
  - export PATH=/cygdrive/c/mingw/bin:$PATH
    ; without, we pick up some cygwin tools at best!
  - cd c:/fptools/fptools
    ; (if you aren't there already)
  - autoreconf
  - ./configure --host=i386-unknown-mingw32 --with-gcc=C:/Mingw/bin/gcc.exe
    ; we use cygwin, but build for windows
  - cp mk/build.mk.sample mk/build.mk
  - in mk/build.mk:
    add line:       SplitObjs = NO
        (MSYS seems slow when there are zillions of object files)
    uncomment line: BuildFlavour = perf
```

```
          (or BuildFlavour = devel, if you are doing development)
      add line:       BIN_DIST=1
    - make 2>&1 | tee make.log
      ; always useful to have a log around

  - Package up binary distribution
    - make binary-dist Project=Ghc 2>&1 | tee make-bin-dist.log
      ; always useful to have a log around
    - cd ghc-6.5
    - chmod +x ../distrib/prep-bin-dist-mingw
      ; if you're happy with the script's contents (*)
    - ../distrib/prep-bin-dist-mingw
      ; then tar up, unpack where wanted, and enjoy
```

Index

Symbols

A

B

C

M

N

O

P

R

S

T

U

W

X