An Introduction to `pconfigure`

Palmer Dabbelt

November 3, 2014

Contents

1 Introduction

pconfigure is designed to be a replacement for GNU autoconf and GNU automake. pconfigure reads a configuration file (called a Configfile, like a Makefile) and generates a Makefile that's suitable for processing by GNU make – in other words, pconfigure fits into the same place in the toolchain that the combination of autoconf, automake, and a run of ./configure does.

Despite this, pconfigure follows a significantly different design philosophy than the GNU tools do: pconfigure is designed to limit the flexibility of the build system, while the GNU tools provide full shell access at any point during the build configuration. pconfigure limits the flexibility of the build system in the hope that pconfigure configuration files from all projects will look somewhat similar and will therefor be easier to understand for users – contrast this to the GNU tools, where each project's build configuration files look very different and are therefor difficult to understand. The general idea is that pconfigure is a build system, while the GNU tools are a framework for creating your own, project-specific build system.

1.1 Building a project that uses pconfigure

If you're just trying to build a project that uses pconfigure, you should be able to run

```
$ cd PROJECT_SOURCE_DIRECTORY/
$ pconfigure
```

and then proceed as if you would when runn the rest of the GNU toolchain. Specifically, running

```
$ make
$ make install
```

will build the code and install it to the default installation directory. Additionally, you can run

```
$ make check
$ ptest
```

to run the project's test cases and print the results of running those test. If you get errors while running pconfigure, Section 6 lists some commen errors and may be useful to help with troubleshooting. Additionally, Section 2 contains a list of command-line options that may be useful to you. A full list of the generated make targets is shown in Section 3.

Note that for a properly written pconfigure project it will not be necessary to run make install before running make check. Some buggy projects will silently run old code in this case, so be sure to consult your project's documentation before attempting that.

1.2 Using pconfigure in your project

In order to use pconfigure in your project you'll need a bit more knowledge about how pconfigure works. The primary means by which you'll be interacting with pconfigure is by modifying it's configuration file. By default, pconfigure will read Configfile in the current working directory, parse every line (until there is an aborting error), and then return. pconfigure configuration files are also refered to as Configfiles, much the same way as make configuration files are referred to as Makefiless.

A Configfile file is line-oriented, and a # character at the beginning of a line signifies a comment, and blank lines are ignored. All other lines are of the format

```
COMMAND OPERATION VALUE
```

where COMMAND is one of the options listed in Sectien 4; OPERATION is one of -=, =, or +=; and VALUE is a free-form text value. If value starts and ends with a ‘ character then it is executed as a shell out (the syntax here is just like BASH and Ruby).

By default, the following configuration files are loaded in this order

```
Configfiles/local
Configfile.local
Configfiles/main
Configfile
```

The general idea is that `*local` are configuration files that should not be checked in to revision control because they're local to a specific instance. This is useful for setting things like compile-time options that you want and other people don't (debug info, for example). The `Configfiles` directory is designed for complicated projects that may have many configuration files, while it's expected that simple projects will just use a singe configuration file and call it `Configfile`.

Note that `pconfigure` will silently skip any of these files that don't exist so you only need to create the ones that make sense for your project. It's not expected that you make either `Configfiles/local` or `Configfile.local` as these will be specific to end-user systems.

1.3 Writing your first `Configfile`

The `Configfile` format is designed to be as terse as possible. In order to achieve this there are a number of implicit rules that `pconfigure` uses to traverse your source tree and attempt to determine exactly how you want your code built. These rules depend on exactly which language is currently being used, a list of which can be found in Section 5.

`pconfigure` was originally designed to link C code. Most of the other backends are at least partially based on the C backend which makes C a good example for how to use `pconfigure`, which is why this example is written using C. The same general structure applies to `Configfiles` for all languages, but each language will handle commands slightly differently so it's important to read Section 5 which will tell you exactly what your language is expected to do.

A simple example `Configfile` for C is shown here

```
LANGUAGES   += c
COMPILEOPTS += -Wall

BINARIES    += hello
SOURCES     += hello.c
```

I'll try to explain each line below.

2 Command-Line Options

The set of command-line arguments that `pconfigure` understands is stored in the `struct clopts` structure and is filled in `clopts_new()`.

2.1 `--config <filename>`

Loads a configuration file, see Section 4.20 for more information. Calling `--config <filename>` is exactly the same as inserting a `CONFIG += <filename>` as the first line of your `Configfile`. Specifically, the configurations will be loaded in the following order:

```
Configfiles/local
Configfile.local
Files listed by --config
Configfiles/main
Configfile
```

2.2 `--version`

Prints the version number of `pconfigure` and exits without touching anything else. Specifically, this doesn't touch the `Makefile` at all.

2.3 `--sourcepath <dirname>`

Allows for the seperation of the source and build directories. By default, `pconfigure` will place output file in the same directory as the rest of your project, this allows them to be seperated. Running

```
$ pconfigure --sourcepath $DIRNAME
```

duplicates the functionality of running

```
$ $DIRNAME/configure
```

in the land of GNU tools.

2.4 `--binname`

See Section 2.6, this is hard to describe in isolation.

2.5 `--testname`

See Section 2.6, this is hard to describe in isolation.

2.6 `--srcname`

The tree commands `--binname`, `--testname`, and `--srcname` are all related and can only really be described together. The idea behind these is to allow interrogating a running `pconfigure` instance from inside the `Configfile`.

The general idea is that you can query the full name of an object that `pconfigure` builds. You must provide either `--binname` or `--testname` whenever you provide `--srcname`. When you pass `--srcname`, rather that outputting a `Makefile` `pconfigure` will print an object name to `stdout`. The binary name will coorespond to the object that gets built for the given source and target name.

For example, if you have the configuration file

```
...
BINARIES += target
SOURCES  += source.c
...
```

then running `pconfigure --binname target --srcname source.c` will produce the object that pconfigure links into `bin/target` when it compiles `src/source.c`. This could be useful if you need to interrogate this output from some script somewhere (for example, from a `GENERATE` script). Note that you can ask for sources that are implicitly included in a binary. The only difference between providing `--testname` and `--binname` is that `--testname` cooresponds to `TESTS` and `--binname` cooresponds to `BINARIES`.

Note that passing any of these options disables `Makefile` output by redirecting writes to `/dev/null`). Additionally, providing any of these arguments will disable any future calls to shell outs, as otherwise a `Configfile` could recurse forever.

3 `make` Targets

3.1 `make all`

Builds every target specified by `BINARIES`, `LIBRARIES`, `HEADERS`, and `LIBEXECS`.

3.2 `make install`

Installs all targets built by `make all` (see Section 3.1) to the specified `PREFIX` path. Note that this may require re-linking some targets in order to deal with something like `-rpath`. Exactly what is rebuilt and how depends on the language, see Section 5 for more information.

3.3 `make check`

Builds and runs every test specified by the `Configfile`.

3.4 `make clean`

Removes every target generated by `make all` (see Section 3.1) and by `make check` (see Section 3.3).

3.5 `make depclean`

Recursively removes `BINDIR`, `OBJDIR`, `LIBDIR`, and `CHECKDIR`. This completely removes every file `pconfigure` could have touched.

3.6 `make uninstall`

Removes every target created by `make install` (see Section 3.2).

4 Commands

4.1 `LANGUAGES += $lang`

First, this flushes the current target.

If `LANGUAGES += $lang` has not yet been run, then $lang is added to the list of known language backends.

$lang is then set as the last added language. This is used for two distinct reasons: to manage exclusive languages, and to set language-global command-line options.

The reason that all languages aren't just added to the language list all the time is because some languages interact in odd ways. An example iss the interaction between C and C++ where C++ can build some C files and C can build some C++ files: it would be impossible to determine which compiler to use. Another example is Scala and Chisel.

The other use of `LANGUAGES` is to set the "last language pointer" so you can modify that language's language-global command-line options. For example, you may have

```
LANGUAGES += c
COMPILEOPTS += option1
...
LANGUAGES += c
COMPILEOPTS += option2
```

when you want to build a whole bunch of targets with `option1`, and then want to build another set of targets with `option2`. Overloading the whole `LANGUAGES` target to both add to the language list and to set a pointer to the current language allows this sort of behavior.

4.2 `BINARIES += $name`

This (and its many variants described below) is the mechanism `pconfigure` uses for tracking targets. When called it first flushes any currently existing target and then sets the current target to `$BINDIR/$name`.

Note that everything except for the target's name is decided when the target is flushed. This includes the target's linker command, which can only be decided based on the source languages that were attached to that target.

When a `BINARIES` target is flushed it will build a staticly-linked executable (as opposed to a static or shared library).

4.3 `LIBRARIES += $name`

Largely the same as `BINARIES += $name` with two major differences: the resulting output binary is stored to `$LIBDIR` instead of `$BINDIR` and a library is built instead of a stand-alone binary.

Some amount of automatic detection is done in order to decide if a shared or a static object should be built. The hurestic is as follows: if $name ends in the canonical shared object extension on Linux for the selected language (`*.so` on Linux, for example) then a shared library will be used, if $name ends in the canonical static archive extension on Linux for the selected language (`*.a` on Linux, for example) then a static archive will be used, otherwise a fatal error will be throw. This mechanism exists to allow for cross-platform portability of `Configfile`s.

Note that `pconfigure` will automatically determine the necessary compiler flags to build shared objects on your platform. For example, Linux and C will result in `-fPIC` being appended to both the compiler and linker flags.

4.4 `LIBEXECS += $name`

Exactly the same as `BINARIES += $name`, except that the resulting output binary is stored to `$LIBEXECDIR` instead of `$BINDIR`.

4.5 `SOURCES += $name`

Adds a `$SRCDIR/$name` to the list of sources that will be linked into the current target. In addition to that source, a number of other sources will be implicitly discovered and linked into the current target. See Section 5 for a discussion of exactly how these dependencies are determined for each language.

Note that you can link a single source file into a target multiple times if it's been compiled with different command-line options. For example, the following snippit

```
BINARIES    += target
SOURCES     += main.c
COMPILEOPTS += -DTEST
SOURCES     += main.c
```

will build `test.c` two times: once with `-DTEST` and once without it. Both of these will be linked into the final executable. Be sure to take this into account when structuring your `#define`s.

Calling `SOURCES` without a target is a fatal error.

4.6 `TESTS += $name`

`TESTS` is the same as `BINARIES` (see Section 4.2), but the resulting binary is built at `make all` and is not installed by `make install`. Instead, it is run at `make check` and the results are collected into `$CHECKDIR`. These results can then be listed by running `ptest` (see Section 7.2).

4.7 `HEADERS += $name`

The `HEADERS` command creates a target that's meant for installing, but it's quite different than the `BINARIES`-based targets. The idea behind `HEADERS` is to allow libraries to install headers into the system. There are two ways headers can be installed: from `$HDRDIR` and from `$SRCDIR`.

Standard practice for C is to build a special set of headers that are installed into the system that contain the public API for a library. `pconfigure` facilitates this by allowing these custom headers to simply be built and installed. The following command

```
HEADERS += libname.h
```

will copy `$HDRDIR/libname.h` into `$PREFIX/$HDRDIR` when `make install` is run, and will do nothing else at any other time.

C++ ABI compatibility necessitates that exactly the same headers are used when the library is built as when it is linked against, which makes writing a custom set of headers pointless. `pconfigure` facilitates this by allowing headers to be copied from other locations at install time. The following snippit

```
HEADERS += libname/class.h
SOURCES += class.h
```

will copy `$SRCDIR/class.h` to `$PREFIX/$HDRDIR/libname` when `make install` is run. Note that in order for this to work you'll have to include the dummy "h" language, which includes a "header compiler" that really just copies the header.

4.8 COMPILEOPTS += $opt

`COMPILEOPTS` is used to change the compile-time options. This maps to the `CFLAGS` variable in the GNU toolchain. `COMPILEOPTS` has different behavior depending on exactly which state `pconfigure` is in:

When `pconfigure` has a current source, then the list of compile-time options for just that source is appended with `$opt`.

When `pconfigure` has no current source but a current target, then the list of compile-time options for all sources linked into that target (both explicitly and implicitly) is append with `$opt`.

When `pconfigure` has no current source or target, then the list of compile-time options for all sources built with the current languages is appended with `$opt`.

When `pconfigure` has no current source, target, or language then a fatal error is thrown.

4.9 LINKOPTS += $opt

`LINKOPTS` is just like `COMPILEOPTS`, but it sets linker options instead of compiler options (essentially it's `LDFLAGS` in the GNU toolchain). See Section 4.8 for a description of `COMPILEOPTS`.

Note that some languages (Chisel, for example) have slightly different semantics as to what's considered a compile-time option and what is considered a link-time option. Check Section 5 for exactly how your language interprets these argumuents.

For most languages (those that use standard compile and link semantics), setting `LINKOPTS` for a source won't do anything, you should instead set it on the target that source will be linked into.

4.10 DEPLIBS += $libname

Adds an internal library dependency to the list of link options. This is essentially the same as calling

```
LINKOPTS += -l$libname
```

but it also adds a `Makefile` dependency such that whenever `lib$libname.so` is rebuilt the target this is attached to will also be rebuilt.

4.11 TESTDEPS += $path

This is similar to `DEPLIBS`, except that it adds an internal test dependency. Note that the format is different: this takes a whole path (as opposed to `DEPLIBS`, which just takes the library name). This difference exists so it's possible to depend on any sort of target.

4.12 `AUTODEPS = bool`

Allows the automatic dependency generation inside `pconfigure` to be disabled for a particular target. The general idea here is that sometimes this won't work: one example is a project with two shared libraries, where one library depends on the other. You really don't want everything to be compiled into the depending shared library, but there's no good way to tell `pconfigure` that.

Luckily that case is simple enough where you really do just want to put *everything* inside a single folder into the library, so it's easy to write a bash script that generates all the `SOURCES` lines and then disable automatic dependency generation all together.

4.13 `COMPILER = $cmd`

Sets the command used to compile for the current language to `$cmd`. Calling this without any language set is a fatal error.

4.14 `LINKER = $cmd`

Sets the command used to link for the current language to `$cmd`. Calling this without any language set is a fatal error.

4.15 `PREFIX = $prefix`

Sets the `$PREFIX` variable to $prefix.

This works in the same manner as the `--prefix=$prefix` argument to GNU's toolchain does: it changes the directory in which built files are expected to be installed. There's no support for things like `--bin-prefix`, instead binaries are installed to `$PREFIX/$BINDIR`. This keeps things consistant between the installed image and the build directory's image.

Note that it's probably not sane to have `PREFIX` in the middle of a `Configfile`. See Section 8.4 for more information.

4.16 `LIBDIR = $val`

Sets the `LIBDIR` variable to `$val`. By default it is set to `lib`.

4.17 `HDRDIR = $val`

Sets the `HDRDIR` variable to `$val`. By default it is set to `include`.

4.18 `TESTDIR = $val`

Sets the `TESTDIR` variable to `$val`. By default it is set to `test`.

4.19 `SRCDIR = $val`

Sets the `SRCDIR` variable to `$val`. By default it is set to `src`.

4.20 `CONFIG += $file`

When `Configfiles/$file` exists and is executable, the output of that executable is treted as a `Configfile` and used as input for `pconfigure`. When the executable terminates, `pconfigure` resumes parsing the current file at the next line.

When `Configfiles/$file` exists and is not executable, that file is parsed as a `Configfile` and used as the input for `pconfigure`. When the executable terminates, `pconfigure` resumes parsing the current file at the next line. This is effectively C's `#include` but for `pconfigure`.

When `Configfiles/$file` does not exist, a fatal error occurs.

4.21 `TESTSRC += $name`

This is just a macro that generates

```
TESTS   += $name
SOURCES += $name
```

4.22 `GENERATE += $name.proc`

Allows for the generation of arbitrary targets. The file `$name` will be generated somewhere in `$OBJDIR`. Some languages (C and C++, for example) will place this directory in the global include path, but the exact semantics are defined by the language. Note that currently it is only legal to call all `GENERATE` commands before any other targets, which results in them being placed in a global directory – this may change in the future such that it is possible to associate `GENERATE` commands with specific targets in order to keep them seperate. `$SRCDIR/$name.proc` is expected to be an executable that has two calling formats:

Calling `$SRCDIR$name.proc --deps` prints out the dependencies of `$name` to `/dev/stdout`. It's up to the user to keep this dependency list complete and to ensure that all these dependencies are eventually handled by `pconfigure`. `pconfigure` provides the `--binname`, `--testname`, and `--srcname` arguments to facilitate this, see Section 2.6 for more information.

Calling `$SRCDIR$name.proc --generate` prints the generated file to `/dev/stdout`. Note that `pconfigure` may call this when `pconfigure` is run in addition to when `make` is run, as opposed to `--deps` which will only be called when `pconfigure` is run (and never when `make` is run). This means it's possible that the script will be run before its dependencies are ready, so plan accordingly!

4.23 `TGENERATE += $script`

Exactly the same as generate, but runs `$TESTDIR/$script` instead of running `$SRCDIR/$script`.

5 Supported Languages

A number of `pconfigure` specifics are determined

5.1 `asm`: Assembly

This is almost exactly the same as C (see Section 5.3), except that it expects source files are called `*.S` and that it compiles using `cc -xassembly`.

It's expected that `cc` accepts the same command-line arguments as GCC does. Note that a wrapper for `nasm` is provided that adapts it to use the GCC command-line arguments. See Section 7.11 for more information.

`asm` supports linking against C.

5.2 `bash`: BASH (`pbashc` compiler)

This uses `pbashc`, a dummy BASH compiler to build BASH code. See Section 7.7 for more information. BASH parses `pbashc #include` directives to discover a list of implicit dependencies.

`bash` supports linking against BASH.

5.3 `c`: C

This is the canonical language. Implicit dependency resolution is handled by parsing C proprocessor macrcos to discover a list of `#include`ed files. This list is then mapped to a list of `C` source files, which are then recursively added.

C supports linking against C.

5.4 `chisel`: Chisel C++ Backend

This language builds a set of Scala files, runs them to procude some C++ code, and then builds that C++ code into an object. Additionally, the generated header file is added to all the other C++ files built. This means that your Chisel code has to come before your C++ code in the `Configfile`.

Chisel implicit dependency detection works in the same manner as Scala (see Section 5.10): an entire directory of files is built and linked together.

Note that specificying a library that ends in `.so` will generate a Chisel library from the generated C++ code, but specifying a library that ends in `.jar` will generate a Scala library directly.

The Chisel C++ backend supports linking against C (for binaries and C++ libraries), and against Scala (for Scala libraries).

5.5 `c++`: C++

C++ is almost exactly the same as C, except that there is a wider set of file extensions is supported (`.c++`, `.cpp`, `.cxx`, and `.c`) and that the code is linked with `c++` instead of `cc`.

Chisel implicit dependency detection works in the same manner as Scala (see Section 5.10): an entire directory of files is built and linked together.

C++ supports linking against C.

5.6 `flo`: Chisel Flo Backend

This language is almost exactly the same as the Chisel C++ language (see Section 5.4) but it generates Flo code instead of C++ code.

The Flo backend supports linking against Flo.

5.7 `h`: C/C++ Header

This is a simple dummy language that just copies header files for installation.

Headers support linking against headers.

5.8 `perl`: Perl (`pperlc` compiler)

This is almost exactly the same as the BASH language (see Section 5.2, but it compiles Perl instead by using `pperlc` (see Section 7.8).

Perl supports linking against perl.

5.9 `pkgconfig`: `pkg-config`

This generates `pkg-config` files by passing them through `sed` a number of times. There's a few rules:

- Most `pconfigure` variables are availiable under the format `@@pconfigure_VARIABLE@@`
- Passing `-Sscript` as a compile option runs `sed` with the contents of that file as an argument
- Passing any other string simply runs `sed` with that as an argument

Note that one `sed` process is run for each argument provided as a compile option.

pkgconfig supports linking against pkgconfig.

5.10 `scala`: Scala

The Scala language uses `pscalac` (see Section 7.9) and `pscalald` (see Section 7.10) to build Scala in a manner that's mostly compatible with a UNIX-style C toolchain.

Implicit dependency handling in the Scala language is handled by assuming that whole directory subtrees are all interdependent and by allowing the Scala compiler to handle all build dependencies. For example, passing

```
BINARIES += target
SOURCES += dir/file.scala
```

will include every scala file listed by `find $SRCNAME/dir -iname *.scala` in `target`. Note that this behavior leads to some added complexity and should be eventually fixed, but it'll be tough because it's hard to parse Scala code in a timely fashion.

Scala supports linking against scala.

6 Common Error Messages

`pconfigure` will print a number of error messages to `stderr`. Some of these messages are fatal and some aren't. Fatal errors result in the termination of `pconfigure` without generating a valid `Makefile`, while non-fatal errors are simply printed to `stderr` while `pconfigure` continues to run. Note that some non-fatal errors may mask or create others, so it's probably best to only rely on the first error message (though this hasn't proved to be a problem yet, it's a big problem with many other languages).

6.1 ... failed, which is probably bad

A shell-out exited with a non-zero exit code. You should avoid using commands that can fail but should instead handle these failures in your code (ie, as opposed to your `Configfile`).

7 Useful Helper Programs

The `pconfigure` distribution includes a number of binaries in addition to the `pconfigure` binary. What follows is a list of these programs along with a short description of their functionality and usage.

7.1 `pclean`

The simplest helper program: essentially it just performs `rm *` recursively. It's just a small wrapper around `find`.

7.2 `ptest`

In addition to being used internally by `pconfigure` to run tests, `ptest` prints the current set of test results to `stdout`. You will almost certainly just want to run `ptest` with no arguments, which will print the entire current set of test results along with a summary.

7.3 pinclude

A small command-line tools that exposes `libpinclude.so`, `pconfigure`'s C preprocessor dependency handler, to BASH scripts. This probably isn't actually useful because both `clang` and `GCC` provide a more robust form of this functionality: the only reason `libpinclude.so` exists in the first place is because it's significantly faster than either of them.

7.4 ppkg-config

A wrapper around `pkg-config` that adds the following command-line options:

- `--optional`: returns success regardless of whether the given `pkg-config` invocation succeeds or fails.
- `--have $name`: adds `-DHAVE_$name` to the compile-time arguments if the given `pkg-config` invocation succeeds, otherwise does nothing.
- `--ccmd $command`: runs a different `pkg-config`-like command. One example is `gpgme-config`. You probably shouldn't be using this to run things like `gcc-config` or `llvm-config`, as their arguments are very different.

These two options combine to allow `pkg-config` to be sufficient to do compile-time detection of features based on what is currently installed on a system. For example, the `pconfigure Configfile` uses the following snippit to decide whether or not to use the system's talloc library or an internal version based on which is availiable.

```
BINARIES    += pconfigure
COMPILEOPTS += `ppkg-config --optional --have TALLOC talloc --cflags`
LINKOPTS    += `ppkg-config --optional --have TALLOC talloc --libs`
SOURCES     += pconfigure/main.c
```

7.5 pgcc-config

This allows `Configfile`s to interrogate the installed C compilers by version number in an attempt to mask flags that are not supported on particular compilers.

7.6 pllvm-config

A wrapper around `llvm-config` that does the same thing as `ppkg-config`.

7.7 pbashc

A bash "compiler", which really just does the following

- Allows `#include` (with the same semantics as in the C compiler) to be used from within BASH files. This is a way to avoid `source`ing BASH files, which can be a pain when you don't know where they'll end up installed.
- Sets the correct shebang at the beginning of the file
- Marks the output as executable

7.8 pperlc

Exactly the same as `pbashc`, but for Perl files.

7.9 `pscalac`

When combined with `pscalald`, this makes Scala look like a regular compiled language. Specifically this wraps `scalac` in a manner such that it produces a JAR archive, much like GCC produces ELF objects.

7.10 `pscalald`

When combined with `pscalac`, this makes Scala look like a regular compiled language. Specifically, this allows the "linking" together of multiple Scala JAR archives into an "executable". Here "linking" means just merging the ZIP files, and "executable" means a self-extracting archive that then runs the result of its extraction (and then cleans up after itself).

7.11 `pnasm`

A wrapper for NASM that makes it take a command-line argument syntax that looks more like what GCC's assembler takes. If you're looking to make NASM a drop-in replacement for GAS in some build script, this may be what you're after.

7.12 `pwinegcc`

A wrapper for the Wine project's modified GCC-based toolchain that makes it behave in a manner more similar to stock GCC. If you're looking to make `winegcc` a drop-in replacement for `gcc`, then this may be what you're after.

8 Odd Behavior (ie, WONTFIX bugs)

`pconfigure` has some odd behaviors. This section attempts to list all of those behaviors and provide ana explination for them. I'd consider them all bugs, but fixing most of them will require imposing some sort of breakage on some `Configfile`s, which I try to avoid.

8.1 Can't link C and C++ into one binary

Building a single binary that consists of both C code compiled by `$(CC)` and C++ code compiled by `$(CXX)` is flaky and has a tendency to break between `pconfigure` releases with pretty much no warning.

This exists because the C++ backend allows C code to be built (as some projects name their C++ source files `*.c`). What this means is that the C++ backend can't determine what is a C++ file and what is a C file by looking at the filename alone.

The accepted workaround is to just build your C code with the C++ compiler when you're trying to link it with other C++ code. This is probably the safest thing to do anyway, as it avoids a bunch of trouble related to C++ name mangling (though you'll have to fix that to build a C library anyway, so it's not a strong argument). This workaround has the unfortunate consequence of requiring that you write C code that can compile with a C++ compiler, which isn't always easy.

8.2 Old Scala classes stick around

Sometimes old Scala class files end up in the build output.

When you build a Scala binary or library, `pconfigure` can't actually determine which class files will be created. Thus, it links every class file that's in the Scala compiler's output directory into the final binary. If you make a source code change that causes a class file to go away, an old version of it will still be in the output directory and will therefor get linked in.

The way to fix this is to "`make distclean; pconfigure; make`", which clears out the whole cache. This isn't ideal, but I'd need to write a proper Scala parser to make this bug go away.

Note that this is kind of a regression: I used to run `strace` to figure out which files the Scala compiler touched, but because `zinc` runs in a background daemon I can't use that trick. I decided it'd be better to have `scalac` and `zinc` produce exactly the same results rather than have `zinc` sometimes break.

8.3 `DEPLIBS` triggers a full rebuild

When you depend on a library that's internal to this project by using `DEPLIBS`, every dependency of that library will be rebuilt whenever `pconfigure` is run.

This is triggered because `pconfigure` uses the library's short name (ie, `lib/libLIB.so` as a dependency of theh binary when you add a `DEPLIBS` command. These short names all depend on `Makefile` because of how the compiler flags hanling works.

Fixing this shouldn't actually be difficult: I believe we could just depend on the full library name instead of the short library name, but the easiest way to implement this would be to make some Configfiles re-entrant that otherwise wouldn't be which will start to break things in tricky ways. A better way would be to store the long library names for later, but that would require that all `LIBRARIES` commands come before their relevant `DEPLIBS` commands, which is probably the sanest way to do it.

8.4 `PREFIX` must be the first line

The `PREFIX` variable is refered to twice: once when the linking step is generated (to set `-rpath`, for example) and once when the `make install` target is generated. It's really best if `$PREFIX` is the same at both times, otherwise unexpected things may happen.