Each recipe consists of 3 main parts: defining identifiers, setting build variables, and defining build commands.

The package “mylib” will be used here as an example

General tips: - mylib_foo is written as $(package)_foo in order to make recipes more similar. - Secondary dependency packages relative to the bitcoin binaries/libraries (i.e. those not in `ALLOWED_LIBRARIES` in `contrib/devtools/symbol-check.py`) don’t need to be shared and should be built statically whenever possible. See below for more details.

Identifiers

Each package is required to define at least these variables:

```
$(package)_version:
Version of the upstream library or program. If there is no version, a
placeholder such as 1.0 can be used.

$(package)_download_path:
Location of the upstream source, without the file-name. Usually http, https
or ftp. Secure transmission options like https should be preferred if
available.

$(package)_file_name:
The upstream source filename available at the download path.

$(package)_sha256_hash:
The sha256 hash of the upstream file
```

These variables are optional:

```
$(package)_build_subdir:
cd to this dir before running configure/build/stage commands.

$(package)_download_file:
The file-name of the upstream source if it differs from how it should be
stored locally. This can be used to avoid storing file-names with strange
characters.

$(package)_dependencies:
Names of any other packages that this one depends on.

$(package)_patches:
Filenames of any patches needed to build the package

$(package)_extra_sources:
Any extra files that will be fetched via $(package)_fetch_cmds. These are
```

```
specified so that they can be fetched and verified via 'make download'.
```

Build Variables:

After defining the main identifiers, build variables may be added or customized before running the build commands. They should be added to a function called $(package)_set_vars. For example:

```
define $(package)_set_vars
...
endef
```

Most variables can be prefixed with the host, architecture, or both, to make the modifications specific to that case. For example:

```
Universal:     $(package)_cc=gcc
Linux only:    $(package)_linux_cc=gcc
x86_64 only:       $(package)_x86_64_cc = gcc
x86_64 linux only: $(package)_x86_64_linux_cc = gcc
```

These variables may be set to override or append their default values.

```
$(package)_cc
$(package)_cxx
$(package)_objc
$(package)_objcxx
$(package)_ar
$(package)_ranlib
$(package)_libtool
$(package)_nm
$(package)_cflags
$(package)_cxxflags
$(package)_ldflags
$(package)_cppflags
$(package)_config_env
$(package)_build_env
$(package)_stage_env
$(package)_build_opts
$(package)_config_opts
```

The *_env variables are used to add environment variables to the respective commands.

Many variables respect a debug/release suffix as well, in order to use them for only the appropriate build config. For example:

```
$(package)_cflags_release = -O3
$(package)_cflags_i686_debug = -g
$(package)_config_opts_release = --disable-debug
```

These will be used in addition to the options that do not specify debug/release. All builds are considered to be release unless DEBUG=1 is set by the user. Other variables may be defined as needed.

Build commands:

For each build, a unique build dir and staging dir are created. For example, `work/build/mylib/1.0-1adac830f6e` and `work/staging/mylib/1.0-1adac830f6e`.

The following build commands are available for each recipe:

```
$(package)_fetch_cmds:
Runs from: build dir
Fetch the source file. If undefined, it will be fetched and verified
against its hash.

$(package)_extract_cmds:
Runs from: build dir
Verify the source file against its hash and extract it. If undefined, the
source is assumed to be a tarball.

$(package)_preprocess_cmds:
Runs from: build dir/$(package)_build_subdir
Preprocess the source as necessary. If undefined, does nothing.

$(package)_config_cmds:
Runs from: build dir/$(package)_build_subdir
Configure the source. If undefined, does nothing.

$(package)_build_cmds:
Runs from: build dir/$(package)_build_subdir
Build the source. If undefined, does nothing.

$(package)_stage_cmds:
Runs from: build dir/$(package)_build_subdir
Stage the build results. If undefined, does nothing.
```

The following variables are available for each recipe:

```
$(1)_staging_dir: package's destination sysroot path
$(1)_staging_prefix_dir: prefix path inside of the package's staging dir
$(1)_extract_dir: path to the package's extracted sources
$(1)_build_dir: path where configure/build/stage commands will be run
$(1)_patch_dir: path where the package's patches (if any) are found
```

Notes on build commands:

For packages built with autotools, ((package)_autoconf) can be used in the configure step to (usually) correctly configure automatically. Any ((package)_config_opts) will be appended.

Most autotools projects can be properly staged using:

```
$(MAKE) DESTDIR=$($(package)_staging_dir) install
```

Build outputs:

In general, the output of a depends package should not contain any libtool archives. Instead, the package should output `.pc` (`pkg-config`) files where possible.

From the Gentoo Wiki entry:

> Libtool pulls in all direct and indirect dependencies into the .la files it creates. This leads to massive overlinking, which is toxic to the Gentoo ecosystem, as it leads to a massive number of unnecessary rebuilds.

Secondary dependencies:

Secondary dependency packages relative to the bitcoin binaries/libraries (i.e. those not in `ALLOWED_LIBRARIES` in `contrib/devtools/symbol-check.py`) don't need to be shared and should be built statically whenever possible. This improves general build reliability as illustrated by the following example:

When linking an executable against a shared library `libprimary` that has its own shared dependency `libsecondary`, we may need to specify the path to `libsecondary` on the link command using the `-rpath/-rpath-link` options, it is not sufficient to just say `libprimary`.

For us, it's much easier to just link a static `libsecondary` into a shared `libprimary`. Especially because in our case, we are linking against a dummy `libprimary` anyway that we'll throw away. We don't care if the end-user has a static or dynamic `libsecondary`, that's not our concern. With a static `libsecondary`, when we need to link `libprimary` into our executable, there's no dependency chain to worry about as `libprimary` has all the symbols.

Build targets:

To build an individual package (useful for debugging), following build targets are available.

```
make ${package}
make ${package}_fetched
make ${package}_extracted
make ${package}_preprocessed
make ${package}_configured
make ${package}_built
make ${package}_staged
make ${package}_postprocessed
make ${package}_cached
make ${package}_cached_checksum
```