# Rust Crate Management in Chrome OS

last update: 2018-02-26, first proposed: 2018-02-26
by: chirantan@

## Objective

Adoption of rust for Chrome OS projects is rapidly increasing. As of today we have crosvm, mosys, and memd. These projects may depend on additional rust crates from the rust community. The goal of this document is to define a plan for managing all of these dependencies in a way that will scale well.

## Background

Rust projects are built using the `cargo` tool. It looks for files named `Cargo.toml` and `Cargo.lock` in the project's top-level directory. These files have a bunch of metadata about the project, including the crates that it depends on. These dependencies are declared using the SemVer specification. This means that a single rust project may depend on multiple versions of the same crate:

- crate foo depends on version 0.3 of crate bar
- crate foo also depends on version 0.2 of crate baz
- version 0.2 of crate baz depends on version 0.2 of crate bar

The `cargo` tool is able to handle these dependencies without issue.

## Requirements

- Keep track of all crates being used in Chrome OS.
- Keep track of licensing and copyright attribution information for third party crates.
- Be able to answer questions like: which packages depend on this rust crate?
- Have an easy way to patch crates for applying security updates.

## Proposed Solution

Put each crate into its own ebuild.

- Built-in mechanism for tracking licensing and copyright attribution information.
- Easily figure out which packages depend on a crate with `equery-$BOARD d <rust_crate>`.
- Apply patches with epatch.

# Implementation Details

## Adding a new crate / updating a crate

Process for adding (or updating) a crate:

- Download a tarball for the appropriate version
- Upload the tarball to the Chrome OS localmirror
- Update the ebuild version / create a new ebuild that matches the version
- Set SLOT="${PV}/${PR}"

## Install source files instead of build artifacts

The rust developers are very aggressive about not maintaining any ABI compatibility between library crates and binaries. As a result there are no meaningful build artifacts that can be installed into a board's sysroot. Instead, we will install all the source files into the board's sysroot and tell cargo (via the cargo eclass) where to find the dependencies. This does mean that the same package may be built multiple times.

Source files will be installed in /build/$BOARD/usr/lib/rust/${P}. We will probably use a new eclass (cros-rust ?) to handle this so that individual ebuilds don't need to manually install files.

## Dependencies go only in DEPEND

As a consequence of installing source files in the board sysroot, having a runtime dependency on a rust crate doesn't really make any sense. Instead all rust crates should only be listed as build-time dependencies.

## Use SLOTs to manage multiple versions of the same crate

Since a single package may depend on multiple versions of the same rust crate we need to provide a way for the source for multiple versions of the crate to be installed at the same time. We will accomplish this using portage SLOTs. Different versions of the package should use different SLOT numbers (ideally SLOT="${PV}/${PR}"). Similarly, the install location for the source files will include the version number in the path so that there are no file conflicts.

## Use SLOT operators for all dependencies

Rust packages should use SLOT operators on all rust crate dependencies. This will ensure that when a crate is updated (and its SLOT is changed) any package that depends on it will get rebuilt.