

CRUX prtpkg Software Maintenance: An Overview

1. Historical Software Maintenance in CRUX—KISS.....	5
1.1. What Is CRUX? What is KISS?.....	5
1.2. In The Beginning, pkgutils.....	5
1.3. Supporting Different Package Configurations: ports, prt-get.....	7
1.4. Official Website For Collections.....	8
2. What prtpkg Adds to CRUX Software Maintenance.....	9
2.1. In a Nutshell.....	9
2.1.1. Shared Maintenance Within or Between Systems.....	9
2.1.2. Maintenance Activity Tracking.....	9
2.1.3. Building Configurations.....	9
2.1.4. Maintenance Policies.....	10
2.2. Rationale.....	10
2.3. What prtpkg Does Not Add to CRUX Software Maintenance.....	11
3. Design Concepts.....	12
3.1. Graphical Overview.....	12
3.2. Introducing Commonwealth: Global and Local Concepts.....	13
3.2.1. Boot and chroot Cells For /etc, /usr, and /var.....	13
3.2.1.1. Concepts For chroot Cells.....	13
3.2.2. Senior and Junior Cells For portdb.....	14
3.2.3. Prime, usrport, and symport Cells For portsu.....	14
3.2.4. Filesystems.....	15
3.2.5. Userids and Groupids.....	16
3.2.6. Environment Variables.....	17
3.2.6.1. BOOTOS.....	17
3.2.6.2. ROOTFS.....	17
3.2.6.3. PRTPKG_CELL.....	17
3.3. Installs and Release Updates For Cells.....	18
3.3.1. Kernel Maintenance.....	18
3.3.2. Installing Into Boot Cells.....	19
3.3.2.1. Pre-existing Commonwealth.....	19
3.3.2.2. New Single-platform Commonwealth.....	19
3.3.2.3. New Multi-platform Commonwealth.....	19
3.3.3. Installing Into chroot Cells.....	20
3.3.4. Release Updates—Overview.....	20
3.3.4.1. Phase 1 for boot Cells.....	20
3.3.4.2. Phase 1 for chroot Cells.....	20

3.3.4.3.	Phase 2 for All Cells (portdb update).....	20
3.4.	Introducing Layers: releases, symports, mixes, builds, and deploys.....	20
3.4.1.	Symport Collection Sets.....	21
3.4.2.	Mixed Cells.....	22
3.5.	Introducing Relationships: porters, builders, and deployers.....	22
3.6.	Introducing Batches: prtpkgbatch and its *.prtpkg files.....	22
3.7.	Introducing New Configuration Files: build.conf and deploy.conf.....	22
3.8.	Introducing /usr/prtpkg and Where To Find Everything.....	22
3.8.1.	/usr/prtpkg/release.....	23
3.8.2.	/usr/prtpkg/broadcasts.....	23
3.8.3.	/usr/prtpkg/builds.....	23
3.8.4.	/usr/prtpkg/cells.....	24
3.8.5.	/usr/prtpkg/groups.....	24
3.8.6.	/usr/prtpkg/mixes.....	24
3.8.7.	/usr/prtpkg/PRTPKG.....	25
3.8.8.	/usr/prtpkg/PORTDB.....	25
3.8.9.	/usr/prtpkg/release/builders.....	25
3.8.10.	/usr/prtpkg/release/deployers.....	25
3.8.11.	/usr/prtpkg/release/distfiles.....	25
3.8.12.	/usr/prtpkg/release/packages.....	25
3.8.13.	/usr/prtpkg/release/work.....	25
3.8.14.	/usr/prtpkg/release/PORTSU.....	25
3.8.15.	/usr/prtpkg/cells/cell.....	25
3.8.16.	/usr/prtpkg/cells/cell/prtpkg.txt.....	26
3.8.17.	/usr/prtpkg/cells/cell/notices.....	26
3.8.18.	/usr/prtpkg/cells/cell/requests.....	26
3.8.19.	/usr/prtpkg/cells/cell/types.....	26
3.9.	Mapping Old Commands Into New Commands.....	26
4.	Processing Organization.....	26
4.1.	Resource Serialization (Locks).....	26
4.1.1.	Serialization Classes.....	27
4.1.2.	Serialization Operations.....	28
4.1.2.1.	lock_obtain.....	28
4.1.2.2.	lock_assume.....	28
4.1.2.3.	lock_freeup.....	28
4.1.2.4.	lock_cancel.....	29
4.1.2.5.	lock_unlock.....	29
4.1.2.6.	lock_giveup.....	29
4.1.3.	Serialization Types.....	29
4.1.3.1.	Global Serialization: PRTPKG.....	29
4.1.3.2.	Driver Config Serialization: PORTDB.....	30
4.1.3.3.	Collection Serialization: PORTSU{* collection}.....	30

4.1.3.3.1.	All collections: PORTSU_*.....	31
4.1.3.3.2.	One collection: PORTSU_ <i>collectionname</i>	31
4.1.3.4.	Port Serialization: MAKE_ <i>portname</i>	31
4.1.3.5.	Build Serialization: WORK_ <i>portname</i> _ <i>version</i> _ <i>buildname</i>	31
4.1.3.6.	Deploy Serialization: PKG_ <i>portname</i> _ <i>version</i> _ <i>buildname</i>	32
4.2.	Inter-process Communication.....	32
4.2.1.	Signal Processing.....	32
4.2.2.	Shared Files.....	32
4.2.2.1.	Broadcast, Request, and Notices Queues.....	32
4.2.2.1.1.	Commonwealth Broadcast Queue.....	32
4.2.2.1.2.	Cell Request Queues.....	33
4.2.2.1.3.	Cell Notices Queues.....	33
5.	Package Components.....	33
6.	Command Information: Help, Prolog, Sample Outputs.....	33
6.1.	Output: prtpkg h [contains TODO items].....	33
6.2.	Output: prtpkg h syntax [contains TODO items].....	34
6.3.	Output: prtpkg h global [contains TOTO items].....	35
6.4.	Output: prtpkg h prt [entirely TODO items].....	35
6.5.	Output: prtpkg h pkg [entirely TODO items].....	37
6.6.	Prolog: prtpkgbatch [contains TODO items].....	37
6.7.	Prolog: prtpkglog [in transition to data reorg].....	38
6.8.	Output: prtpkglog [in transition to data reorg].....	38
6.9.	Output: cat /usr/prtpkg/cells/dlcz[ZD]/CRUX-3.2/pkgsb3/log/00031/06.log.....	38
6.10.	Output: cat /usr/prtpkg/CRUX-3.2/prtpkg_by_col.20170130-161126.log.....	41
6.11.	Output: pkg_basenames [in transition to data reorg].....	41
6.12.	Prolog: localize_ports [in transition to data reorg].....	42
6.13.	Prolog: missing_packages [in transition to data reorg].....	42
6.14.	Prolog: missing_packages_doit (gawk) [in transition to data reorg].....	42
6.15.	Output: misspkglog [in transition to data reorg].....	42
6.16.	Prolog: pkgaddconf [targets contain TODO items].....	42
6.17.	Prolog: prtlist [in transition to data reorg].....	43
6.18.	Prolog: prtlist_packages (gawk) [in transition to data reorg].....	43
6.19.	Output: prtlist [in transition to data reorg].....	43
6.20.	Prolog: prtpkg_symlink (gawk).....	44
6.21.	Prolog: prtpkginfo [in transition to data reorg].....	44
6.22.	Output: prtpkginfo -h [in transition to data reorg].....	44
6.23.	Prolog: validate_builds [very early new program].....	44
6.24.	Prolog: validate_symports [in transition from varports].....	45
6.25.	Prolog: validate_symports_links (gawk) [in transition from varports].....	45
6.26.	Prolog: whatpkg [in transition to WHATPKG_ variables].....	45

6.27. Prolog: whatpkg_2ndline (gawk) [in transition to WHATPKG_ variables].....	45
6.28. Prolog: whatpkg_pkginfo (gawk) [in transition to WHATPKG_ variables].....	45
6.29. Prolog: whatpkg_prtpkgtxt (gawk) [in transition to WHATPKG_ variables]...	46
6.30. Prolog: whatprt [in transition to WHATPRT_ variables].....	46
6.31. Prolog: whatprt_doit (gawk) [in transition to WHATPRT_ variables].....	46

1. Historical Software Maintenance in CRUX—KISS

1.1. What Is CRUX? What is KISS?

[CRUX](#) is a source-based Linux distribution that lists the KISS (Keep It Simple, Stupid) design concept as its primary design goal. While these days that can seem highly unorthodox for a Linux distribution, it **is** the core of the UNIX Philosophy as described by Eric S. Raymond in [The Art of UNIX Programming](#) (specifically [The Unix Philosophy in One Lesson](#) chapter).

A highly significant derived design goal for CRUX has emerged in recent years—[systemd](#) has no place in a CRUX *system*ⁱ. That amazingly popular alternative to the idea of the simple *init* daemon, a feature of UNIX from its early years (the relatively new [sysvinit](#) package originated ~1980), is staunchly considered egregiously flaunting of the UNIX Philosophy by CRUX developers. See the other chapters of Raymond's treatise for the important details.

The KISS principle has been applied to CRUX software maintenance from its inception. For instance, CRUX does not repackage or even *install* the kernel. Except for providing kernel source and a suitable `.config` file (see Section 3.3 for details), CRUX *ignores* kernel maintenance, leaving it to the sysadmin to decide what to do according to the kernel's README file and [kernel.org](#) announcements.

1.2. In The Beginning, pkgutils

In the early days of CRUX, founding developer Per Liden created the heart of CRUX software maintenance, the `pkgutils` package that facilitates automating the standard procedure for introducing FOSS packages into a system:

1. *download* the package source code [tarball\(s\)](#),
2. *extract* the tarball content into a usually temporary build (aka work) directory,
3. *configure* the build directory for building usually via a ***configure*** script,
4. *build* (compile and link, etc.) the package using a ***make*** command, and
5. *install* the built results into the system, usually using a ***make install*** command.

This final step must be performed by the *superuser* who is responsible for the availability and integrity of the platform.

With `pkgutils`, the entirety of the standard procedure is within the purview of the `pkgmk` command with a twist: the final phase does not “install” the package into the system doing the building; i.e., the root filesystem *tree*ⁱⁱ. Instead, an alternative tree is defined, usually the build directory itself, via a `DESTDIR=` parameter supported by the `make install` configuration for the package. The final task of `pkgmk` processing, which is outside the standard procedure, is to create a CRUX *package file* (aka *package tarball*) that is intended to be extracted into the real root tree of the target system by the superuser. A security advantage of this approach to `make install` is that full superuser authority is not necessary. Instead, the `fakroot` package can be used to define privileged *inode* metadata in a tar, not in the logical root filesystem, without allowing any superuser authority. CRUX encourages using `fakroot` with a no-login user for *everything* `pkgmk` does. A foundational concept of `pkgutils` maintenance, then, is building binary executables from source must be separate from installing them into a system, and that the build results should be encapsulated in a binary-code package file that can be copied to and installed into a suitable CRUX platform without requiring the source first be built on that platform.

The `pkgadd` command of the `pkgutils` package is run to install/update CRUX package files into the running or mounted target system, which of course is only permitted for that system’s superuser. Uninstalling CRUX package files is the job of the `pkgrm` command. Both manipulate the `/var/lib/pkg/db` flat file that contains the inventory of packages installed on that system, including their versions and all files installed on their behalf. Use `rejmerge` to resolve file update rejections.

Note the `pkgutils` methodology remains fundamental to CRUX software maintenance, and has not changed dramatically for almost two decades. However, it is completely ambivalent regarding just what it is maintaining. `Pkgmk` learns nothing outside the *package directory* in which it is invoked save its configuration as defined via command parameters and a configuration file. It does not even care to know the path of the package directory. Also called a *port*, the package directory contains all the package-specific data `pkgmk` needs to do its job except the actual source tarballs involved. The port’s `Pkgfile` text file is central. In fact it is sourceable `bash` code (not directly executable) that contains a `build()` function that is invoked by `pkgmk`, itself a fully executable `bash` script, to perform steps 3–5 of the standard procedure (not including the actual installation into a platform as

previously discussed). CRUX developers still highly esteem pkgutils' aloofness from package metadata and its abhorrence of activity logging.

Thus, pkgutils is of next to no use when it comes to distinguishing multiple ports of any particular package from each other and selecting the ports to be processed by pkgutils.

1.3. Supporting Different Package Configurations: ports, prt-get

CRUX had grown to the point of organizing package ports with similar characteristics together into directories of ports called *collections*. The officially supported core, opt, compat-32, xorg, and quasi-supported contrib collections are simply divisions of the packages considered canonical to CRUX—there is only one port per package in this set of collections.

The **ports** command (and package) was introduced by Per Liden to assist with maintenance of entire collections. Its biggest feature is provided via the **-u** flag, which causes updating the specific (or by default all activated) local mirror collections with any changes made to the master collections available via the Internet. Thus, collection maintenance in CRUX is undertaken on a pull basis. Also, the **rsync** model is used, so local collection mirrors have ports deleted when their Internet counterparts are deleted. Consequently, only the known current version of any port is ever available in any particular collection, or not—mirrored collections alone can contain back-level or dropped ports, but only as long as they remain unsynchronized with their masters.

However, as expected, folks who installed CRUX on their computers often found they had a need to modify the canonical port of a package in order to accommodate capability requirements incongruous with CRUX canon. The solution was and is to fork the canonical port, copying it (**cp -a**) into a *private collection*, modifying its Pkgfile and anything else in the port as needed, and using the pkgutils commands on that port instead of the canonical port. Private collections were (and are) named according to the sysadmin's fancy as long as they didn't conflict with any other collection's name, and these began to be shared with other CRUXers via public *repositories* (just another word for collections). Port management (packages are not static and neither are their ports) was increasingly a chore for everybody.

In response to that itch, Johannes Winkelmann scratched out the next layer of CRUX software management infrastructure from inspiration based in the Debian distribution's apt-get facility. The **prt-get** command uses a configuration file of

important options to control its processing, especially the `prtdir` statements that (1) describe what local collections the tool knows about and (2) define how one of many available ports for a particular package is selected for processing. According to the specific ***prt-get*** subcommand the maintainer issues, the utility can invoke ***pkgmk***, ***pkgadd***, and/or ***pkgrm*** as needed to bring about that ***prt-get*** subcommand's objective. In many cases, `pkgutils` commands were no longer used directly to routinely perform CRUX software maintenance.

The ***prt-get*** subcommands ***sysup***, ***depinst***, and ***grpinst*** in particular greatly simplified software maintenance by supporting a new wrinkle, *dependency handling*, another important feature that became canonical to CRUX like `prt-get` itself. The format of the `Pkgfile` was extended to allow inclusion of a `# Depends on:` statement that specifies a list of blank-separated packages that need to be installed to build and/or run the package having the statement. Although it is much more simple than other distributions' dependency capabilities, and not always correct as a consequence (usually when non-canonical ports are involved), it serves as a crucial starting point for identifying the order in which packages need to be processed.

There have been other extensions, some canonical. An inadequately documented facility to *alias* depended-upon package names was added to `prt-get` to provide a rudimentary *package provides generic package* and *package depends on generic package* capacity. The `/var/lib/pkg/prt-get.aliases` flat file, which may be customized as needed by an installation, defines such relationships of the first kind. Also, some CRUXers use the `opt/mpup` package to help manage forked ports—see its README file for the details (`prt pkg` does not yet directly support that software). Very recently onodera announced a [new ports/prt-get work-alike tool](https://crux.nu/portdb) on `#crux`. While I have not yet had the opportunity to deeply look into it, it is my expectation it can be fit into the `prt pkg` framework without too much difficulty.

1.4. Official Website For Collections

<https://crux.nu/portdb> is the canonical resource for all known CRUX collections, particularly for downloading the `/etc/ports` *driver config* files for the collections, which are read by ***ports -u*** to access the collections' master repositories. Changes to the set of master repositories and the contents of driver config files usually occur unannounced. In addition, the site maintains only one release and such transitions can also occur without notice, although one can be expected within a month following the announcement of a new CRUX release. All

the canonical collections are versioned. Non-canonical collections can change between versioned and unversioned at any time, again, often silently.

2. What prt pkg Adds to CRUX Software Maintenance

Prt pkg enhances port and package maintenance by providing the following missing pieces in a standardized and hopefully easily adaptable manner for any enterprise deploying one or more CRUX platforms:

2.1. In a Nutshell

2.1.1. Shared Maintenance Within or Between Systems

Prt pkg supports multiple distribution releases on a single system and/or supports multiple systems, allowing designated systems to provide prt pkg services: crux.nu/portdb synchronization (portdb), collection synchronization (portsu), package building (prt pkg), and basic pkgutils (deploy) service (defined for now as self-service). This includes tracking what specific systems and groups of systems are running what releases and are served by what servers. Such a configuration of multiple releases and/or systems sharing a common prt pkg maintenance infrastructure will be referred to as a *commonwealth*.

2.1.2. Maintenance Activity Tracking

Prt pkg supports maintenance logging, especially the ability to *know* what *ports* are currently installed, not just what *packages* are installed; plus true chronological logging that facilitates easy commenting upon the maintenance process directly in the logs produced (as [*What Mother Never Told You About VM Software Service*](#) clearly stated in March 1983, “Rule Number Four: Always Leave Tracks”). This includes taking snapshots into a compressed tarball of the build environment for *pkgmk* including the package directory tree, currently installed ports (implicitly pointing to their build environment snapshots), the kernel’s config file, etc.

2.1.3. Building Configurations

Prt pkg supports multiple building configurations, enabling you to both define and know exactly *how* a port will be/was built; e.g., PATH and CFLAG settings inherited by and possibly not overridden by *pkgmk* or the package being processed. This facilitates (1) designating what “builddefs” you want to permit for any set of ports and/or systems within the commonwealth, and (2) selecting the builddef variety

(prtpkg provides default and debug, or use your own) of any or all packages you wish to deploy on any CRUX system.

2.1.4. Maintenance Policies

This grouping of capabilities deals with defining the way software maintenance is done and not done at the enterprise level and down to the level of individual systems, and implementing protections against software maintenance contrary to such policies (to help you avoid shooting yourself in the foot or worse—at the enterprise level such disasters can be, well, disastrous).

2.2. Rationale

Together, these capabilities could be reasonably hailed as industrial-strength CRUX software maintenance. They set the stage for enterprise-wide automated maintenance roll-outs as well as cross-architectural package builds in the future.

The state-of-the-art for maintaining CRUX is geared to one or two completely independent platforms. The concept of an organization maintaining hundreds (let alone thousands) of CRUX hosts is a dream, or more likely a nightmare, for CRUX developers. At this time it does seem the probability of an enterprise needing a team of CRUX sysadmins is microscopic. However, as appealing to an organization as CRUX might be vis-à-vis most Linux distributions, the problems of scaling canonical software maintenance make the proposition too expensive to pursue. Thus, canonical maintenance is limiting much needed wider acceptance of the distribution, which in turn limits CRUX' mindshare and inhibits expanding the microscopic pool of developer resources.

The challenge for prtpkg design is envisioning a one-size-fits-all architecture that can be coded as needed while delivering a subset most likely to be useful to a wide center of the bell curve of potential users. The KISS concept becomes a trifle unclear as new capabilities are added to engineering systems—it seems the system is becoming less simple. First and foremost, new tricks must have compelling justification to be incorporated—there *must* be a crying need for the features within a significant subset of the installed base. Once past that requirement, the design must be straight-forward, clear, near-universal in applicability, easily configured and customized... well, what ESR wrote already. Perhaps this should be termed KIASAPS (Keep It As Simple As Possible, Smartie). But in reality it is part and parcel to KISS, just not so obvious. Simple is not always orthogonal to complex (Exhibit A: the current Linux kernel vis-à-vis version 0.01 released

September 17, 1991). Making simplicity a higher goal than ensuring security, integrity, and perhaps maintainability is unwise.

It may be prtpkg will never become canonical. That is fine. It may even be prtpkg will never be utilized by anyone besides its originator. That, however, is not fine, as previously discussed. But either way, folks should be aware it is available, what it does, and how. If you're a CRUX developer, please avoid breaking it beyond repair if at all possible. If there is sufficient interest, I am willing to pursue establishing a git repo before putting up my first port of the prtpkg package (I still need to finish the first releasable source tarball!), which would most likely compel me to port this still incomplete document to some other format.

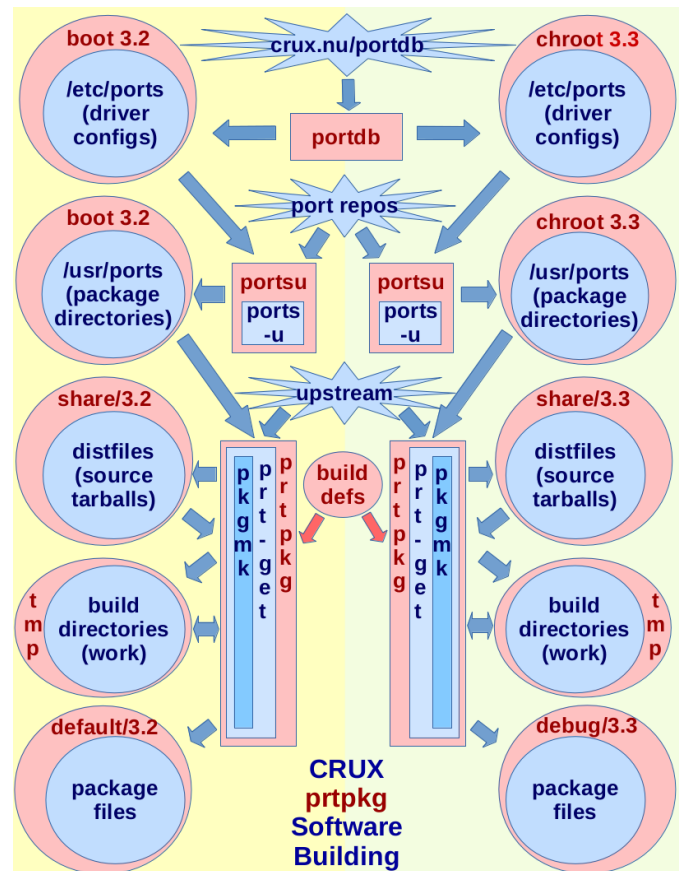
2.3. What prtpkg Does Not Add to CRUX Software Maintenance

Note prtpkg does *not* require any changes to any canonical packages, with one tiny post-**pkgadd** exception: the software maintenance command executables are renamed to hide them from anything that would directly invoke them. In their places prtpkg substitutes executables that filter out with suitable error messages any requests that are not permitted because they (1) are contraindicated by the maintenance policies of the enterprise or (2) could produce changes that corrupt the prtpkg view of reality (not to mention could be unknown to you, the system administrator). If the request is harmless, the front-ending code logs any non-prtpkg invocation and passes the request (via `execvr(2)` or the **dash** built-in command **exec** as is appropriate) to the renamed original unmodified **ports**, **prt-get**, **prt-cache**, **pkgmk**, **pkgadd**, or **pkgrm** executable for normal processing. It is likely **mpup** will be added to this list as well. Work on these front-ends has yet to commence.

3. Design Concepts

3.1. Graphical Overview

This graphic diagrams how canonical CRUX software building works (blue objects) and what prt pkg adds (red objects). The rectangles are processes (some nested), the ellipses are data areas in the system, and the stars are Internet resources (imagine you're looking down at them hanging over the boxes and ellipses, downloading data into the boxes). The **portdb** command at the top updates /etc/ports trees for all affected releases (the canonical approach requires the sysadmin to do this manually if at all). Similarly, the **portsu** commands, invoking **ports -u**



commands, manage the release-specific /usr/ports trees. The **prt pkg** commands ride herd on **p r t - g e t** commands that **cd** into the /usr/ports/collection/port directories and launch **p k g m k** commands that download the upstream distfiles, set up the work directories, expand the source tarballs, then build the packages and “install” them into CRUX package files. What the canonical tools don’t know is the **prt pkg** commands set things up according to the enterprise’s desired build definitions and that the package files are being stored in locations that show exactly how and from what they were built.

If you consider the diagram split vertically down the middle, you see two different build *cells*ⁱⁱⁱ, but there can be more cells in the commonwealth, including boot and chroot cells on different hardware platforms both real and virtual. The associations shown could be 3.3 boot and 3.2 chroot instead, as this is only an illustration. Notice from the canonical perspective, the two cells are close to identical and are totally unaware of each other’s existence—the only differences are the versioned collections and ports they process and the likely different *cell* components, especially versioned and *builddefed* build tools within the *cells*’ PATH definitions. Without prt pkg involvement, this diagram requires two bootable CRUX systems or a lot of programming and system administration effort.

3.2. Introducing Commonwealth: Global and Local Concepts

To permit a single prt pkg platform to be a prt pkg *porter*^{iv}, *builder*^v, or *deployer*^{vi} for multiple *releases*, and in consideration of the **ports -u** command's requirement that the /etc/ports and /usr/ports trees of the system running it contain the data it is to process, prt pkg is designed to use a **chroot** approach for maintaining distribution releases other than that deployed in the prt pkg platform's boot cell. While initially prt pkg will not explicitly support prt pkg platforms that are virtual machines or containers, it may come to light prt pkg really cannot distinguish such from real prt pkg platforms and so supports them inherently. The take-away point of this paragraph is prt pkg is designed to support cells, not systems; i.e., there's not much difference to prt pkg between a system and a cell unless the system is not also a cell (then it just isn't of interest). When a cell deploys prt get-built packages within itself, that should not interfere with any system persona the cell might have, just as that is true for canonical pkgutils activities within a non-prt pkg CRUX system (/etc/pkgadd.conf and **rejmerge** are your friends). However, understand such system administration concerns are outside the scope of the prt pkg package documentation.

3.2.1. Boot and chroot Cells For /etc, /usr, and /var

Whether /etc, /usr, and /var reside in the boot cell's root trunk or in a chroot jail elsewhere in boot cell's tree, prt pkg considers these construction sites egalitarianly and calls them both cells, jail or not (prt pkg chroot jails contain only one cell).

3.2.1.1. Concepts For chroot Cells

The initial prt pkg chroot design concepts are:

1. prt pkg chroot cells do not have a system persona—they do prt pkg work exclusively (non-prt pkg CRUX systems booted and chrooted are of course free to perform canonical software maintenance including simply running **pkgadd** using prt pkg-built packages without the benefit of prt pkg tracking and serialization safeguards);
2. prt pkg chroot cells are treated the same as real machine boot cells, virtual machine and container boot cells, and any virtual systems' chroot cells; and
3. prt pkg **chroot** commands use only local chroot cells (this restriction may be relaxed if testing reveals no logical or performance impediments when chrooting into cells accessed via network filesystems).

3.2.2. Senior and Junior Cells For *portdb*

Cells are also distinguished by their proximity to the *portdb* process. Those that are designated as senior cells by the enterprise have their `/etc/ports` tree directly updated by *portdb*, which is designed to closely mirror `crux.nu/portdb` by updating at least once every 24 hours under control of a `crontab` entry. On the other hand, the `/etc/ports` trees of junior cells are updated according to the cells' update policies as defined and implemented by the sysadmin; e.g., when the phase of the moon is just right, examine the state of the associated senior cell's `/etc/ports` tree and update the junior cell's `/etc/ports` contents with their senior counterparts by creating and running a well-commented *prtpkgbatch* file of one or more *cp -p* and/or *rm* commands, thereby leaving excellent tracks. Consider senior cells as subscribed to *portdb* updates and junior cells as not. The prtpkg design requires senior cells to fully mirror all `/etc/ports` content of which `crux.nu/portdb` is cognizant. While prtpkg fully supports the retention in senior and junior `/etc/ports` trees of driver config files that are no longer available from `crux.nu/portdb`, such retention is not a requirement. There must be at least one senior cell in a commonwealth for every distribution *release* therein.

3.2.3. Prime, *usrport*, and *symport* Cells For *portsu*

Lastly, cells are also distinguished by their proximity to the *portsu* process. Those that are designated as prime cells by the enterprise have their `/usr/ports` tree directly updated by *portsu*, which is designed to closely mirror the repositories identified in the active driver config files of the cell's `/etc/ports` tree, by updating at least once every 24 hours under control of a `crontab` entry. On the other hand, the `/usr/ports` trees of *usrport* and *symport* cells are never so updated, but only according to the cells' update policies as defined and implemented by the sysadmin. Consider prime cells as subscribed to *portsu* updates and *usrport* and *symport* cells as not. While *usrport* cells can be updated using *portsu* or even *ports -u* directly, that is not required and in some situations using *ports -u* directly may be inadvisable. There need not be `prtdir` statements for every active collection in the prime cell's `/etc/prt-get.conf` file but every commonwealth probably ought to have at least one prime cell for each included *release* that does. To prevent *portsu* from unknowingly updating the same port multiple times, *symport* layouts (see Section 3.4.1 for the explanation) must not be used in prime cells; *symport* layouts are only supported for *symport* cells. While it is anticipated prime cells will be senior cells as well, that is not required. There must be at least one prime cell in a commonwealth for every distribution *release* therein. Since *symport*

cells link into a particular prime or usrport cell, an entanglement of these cells is recognized (see Section 3.4.2 for details).

3.2.4. Filesystems

In a multi-prtpkg-platform commonwealth, some of the data managed can and must be shared between cells via a [network filesystem](#) and some, like the `/etc/*.conf` files and the `/var` tree, cannot. Well, they could if they could be... well, a lot of things, but it's really easier to just keep them local than attempting to make them sharable. It's a similar problem to supporting multiple releases on one system which is solved using chroot cells. In fact chroot cells should be shareable across systems with the proper serializations, but in many cases network overhead and propagation delays can render the performance inadequate for things like rebuilding firefox or qt5.

There are several network filesystem packages that are well-supported on Linux distributions. None are trivial to plan, establish, and maintain. [NFS](#) is CRUX-canonical (see `opt/nfs-utils`) and is installable via the ISO distribution. [Samba](#) is the other canonical network filesystem (see `opt/samba`), but it's only for masochists or those who work for a sadist (it should be workable for prtpkg if you can control [mangling](#) and [security](#) adequately). If you're willing and able to port the software, [AFS](#) is great if you can swallow its Kerberos dependency, [Ceph](#) is up and coming (though for now its CephFS component is still beta quality), and the Plan 9 paradigm called [9P](#) developed by [Ritchie and Pike](#) might be a good fit for your enterprise's requirements. The point here is if prtpkg is your enterprise's first foray into shared filesystems, you really need to look at the big picture for filesystem sharing, not just what prtpkg requires, to ensure the effort expended provides the greatest benefit. It's better to get it right the first time (yes, there's "Rule Number Three: Don't Expect It To Work" but that is only in regard to always having a dependable backout plan at the ready when making any system change).

All that prtpkg wants from a network filesystem is the fundamentals—enable the cells in the commonwealth to cooperatively share a common file namespace. It does not even require that filesystem provide intrinsic locking support (see Section 4.1 for details about the prtpkg resource serialization facility). Of course, prtpkg locks should work just fine on networking filesystems that have their own locking mechanisms, so prtpkg can probably just ignore any such filesystem locking.

The location of senior and prime cells needs to be carefully considered. Because they are directly updated by *portdb* and *portsu* respectively, they need to be exported into shared namespace if those processes are not running in those cells. Cells' *prtpkg* processes do need to be local to the cell, though, for performance considerations. Since it is usually better to avoid exporting boot cells' root filesystems, senior and prime cells should probably be limited to chroot cells if their *portdb* and *portsu* processes will run remotely from them. It's probably better for senior and prime cells to not have a system persona, as well. Considerations may indicate one or more boot cells' root filesystems need to be exported into the network filesystem tree, despite best practices. [Security risks](#) may be acceptable. Network performance may not be a bottleneck. You decide.

For now, prtpkg is not testing core/acl and selinux configurations. Let us know how that goes if you venture forth so prtpkg can better support everyone.

An unscheduled outage of the shared commonwealth components, especially the lock directories, is a critical problem if any update locks other than the PRTPKG lock are in granted state at the time of the outage. These resources should be deployed on the highest-availability system(s) possible to make such difficult recovery tasks as unlikely as possible. Each cell in the commonwealth needs to maintain a PRTPKG state indicator in a local filesystem indicating if the shared prtpkg infrastructure is functioning or not using a 60-second watchdog function, possibly launched via crontab. The craziness that can ensue from losing the shared filesystem is another reason not to *chroot* into remote cells.

3.2.5. Userids and Groupids

In addition to the pkgmk userid and nobody groupid common to canonical CRUX software maintenance (that prtpkg requires), a prtpkg userid and groupid are recommended to facilitate limiting superuser authority available during software maintenance within a commonwealth. The userid should have limited *sudo* authorization to minimize risk (but prtpkg does not require sudo), and the groupid should act the same as wheel commonly does for root (or just use wheel).

Of course, in a commonwealth, all userids and groupids need to have consistent uids and gids across cells. While opt/openldap is canonical, it might be simpler to develop scripts to perform the needed maintenance in each cell using the request queue facility described in Section 4.2.

3.2.6. Environment Variables

3.2.6.1. BOOTOS

A boot cell's `/etc/ports`, `/usr/ports`, `/var/log`, and `/var/lib/pkg` trees and `/etc/ports/pkgmk.conf`, `/etc/ports/pkgadd.conf`, and `/etc/ports/prt-get.conf` files are associated with the distribution *release* the cell last booted. Those trees in a chroot cell are likewise associated with the distribution *release* installed in the cell. That release is defined in the `BOOTOS` environment variable added to `/etc/rc.conf` file. For a boot cell it is automatically exported to every new session by an addition to `/etc/rc`, but for a **chroot** session it may need to be reexported as part of the session's initialization.

3.2.6.2. ROOTFS

Also new to `/etc/rc.conf` is the `ROOTFS` environment variable, the filesystem identifier of the cell's root filesystem.

For cells using a filesystem as their root filesystem (true for all boot cells), prtpkg expects the `ROOTFS` string will be maintained as the name of a file in the root filesystem's trunk, having the form `[label]`; e.g., `[XY]`. It is recommended that *label* (without the enclosing brackets—`mount -l` shows any such label in brackets) be in fact the partition's, volume's, and/or filesystem's label (see the `mkfs_XX` script for a template to build an ext4 partition accordingly).

For chroot cells not using their own mounted filesystem, it is required the chroot's trunk contain a `ROOTFS` file that shows the *label* is not a filesystem label; e.g., `[SSD1_xyzzy]` where the `SSD1` is the label of the filesystem housing the chroot tree. This string is defined as `ROOTFS` in the chrooted `/etc/rc.conf`.

This filesystem identification convention requires the `ROOTFS` file is the only file in its directory conforming to that format; i.e., starting with a left bracket and ending with a right bracket.

For chroot cells, `ROOTFS` must be exported to the environment as part of the **chroot** session's initialization since a **chroot** session does not boot up.

3.2.6.3. PRTPKG_CELL

The `PRTPKG_CELL` variable uniquely identifies the local cell for prtpkg purposes. It is simply the concatenation of the cell's `/etc/rc.conf` `HOSTNAME` value (returned by the `hostname` command) and the cell's `ROOTFS` value (see Section 3.2.6.2); e.g.,

server1[HDD2], server4[SSD3_CRUX-3.4_prime]. PRTPKG_CELL is very nice in a PS1 prompt.

3.3. Installs and Release Updates For Cells

The canonical approach to installation or update of a CRUX distribution release uses (for update) a scheduled outage of the target system (for install that's unnecessary) during which the release's ISO is booted and the system is installed or updated as needed. Within a prt pkg commonwealth, this canonical processing is expanded, mostly via a merge and enhancement of the canonical **setup** scripts into the **prt pkg_setup** script, plus the new **mk_cell** and **up_cell** scripts (for chroot cells only that need not the ISO environment). One of several paths is taken according to the type of commonwealth involved and commonwealth processing can precede and/or follow the canonical package installation phase of script's activity.

Other differences between canonical and prt pkg installation/updating should be noted. If **prt pkg_setup** discovers a prt pkg root filesystem mounted as /mnt, it assumes that is the target and does not ask for the target's location. Also, **prt pkg_setup** asks for the host name of the new prt pkg platform while **setup** defers that to editing /etc/rc.conf in the **chroot** phase of the install.

3.3.1. Kernel Maintenance

The ISO boots a recent stable Linux kernel built using the ISO's .config file which the ISO contains along with the vanilla kernel source tarball that kernel was built from plus any applicable vanilla patches. The ISO's **setup** script looks in the target's /usr/src tree for the ISO kernel's version. If it's not there, (1) the kernel tarball is expanded into the target's /usr/src tree, (2) any kernel patches the ISO contains are applied, (3) the ISO's default .config file is copied into the new tree, and (4) if the target's /lib/modules tree for the release does not exist, it is created and the target system's module dependencies are pre-populated via a **depmod -b \$ROOT -a \$KERNEL_VERSION** command. Other than this, CRUX completely ignores kernel maintenance. The release-dependent [CRUX Handbook](#) expects the sysadmin installing or updating the system will **chroot** into the target root filesystem from the ISO system and configure, build, and install a kernel therefrom (not necessarily the one the ISO may have expanded). CRUX does not provide, let alone maintain, a kernel package that can be processed by pkgutils. As most CRUXers cannot wait for a new CRUX release every 12-18 months to update their kernels, they do what the CRUX developers expect: they retrieve

kernel releases and/or patches from kernel.org, expand/apply them, customize the .config files as needed by the enterprise and the platforms, and build and install the new kernels, all according to the kernel's README document, just as they did for the install or upgrade. CRUX development neither adds to kernel distributions nor engineers CRUX-specific kernel patches.

All of this is also true for prtpkg software maintenance systems; however, an enterprise is completely at liberty to add its own kernel software maintenance policy and automata to its own commonwealth(s). Contribution of same to the community is certainly encouraged—there will hopefully be many organizations with similar goals in this regard.

3.3.2. Installing Into Boot Cells

Installs into boot cells begin in the canonical way by booting the release ISO on the not-yet-a-cell system.

3.3.2.1. Pre-existing Commonwealth

For installs involving a pre-existing commonwealth, some sysadmin legerdemain installs any needed networking filesystem software into the RAM-resident root tree, configures it, and mounts all needed network filesystems into the target root tree, usually mounted as /mnt in the RAM-resident root tree, before running the **prtpkg_setup** script instead of the canonical **setup** script to convert the freshly formatted target root tree into a new boot cell root tree. This includes obtaining the PRTPKG update lock to safely add the cell to the commonwealth (refer to Section 4.1.3.1 for details). Experience with installing new platforms into existing commonwealths may lead to revisions in this approach that will ease the complexity of the legerdemain phase, possibly with goodly amounts of prtpkg package and/or homegrown automation.

3.3.2.2. New Single-platform Commonwealth

If a single-prtpkg-platform commonwealth is being co-installed with the boot cell, a /usr/prtpkg tree is simply established in the target cell's root tree.

3.3.2.3. New Multi-platform Commonwealth

However, if a multi-prtpkg-platform commonwealth is being co-installed (only NFS is installable from the ISO and must be installed for this situation), **prtpkg_setup** configures the boot cell's network filesystem server to export its /usr/prtpkg tree per a dialog with the sysadmin (that may prove to involve homegrown scripts).

3.3.3. Installing Into chroot Cells

For new chroot cells, installation is accomplished by running the *mk_cell* script in the associated boot cell as a normal command without needing to reboot anything (TODO really soon). Such installs are only supported from boot cells already within a commonwealth.

3.3.4. Release Updates—Overview

Updates are only performed for cells already within a commonwealth. There are two aspects of the process regardless of the cell types: (1) the update of the core toolchain packages and the BOOTOS in the `/etc/rc.conf` file, and (2) the update of versioned collections to the new version; e.g., in file `/etc/ports/core.rsycn`, the line `collection=ports/crux-3.2/core/` becomes `collection=ports/crux-3.3/core/`.

3.3.4.1. Phase 1 for boot Cells

For boot cells, the first phase is effected by booting the versioned CRUX ISO in the cell being updated and using the *prt pkg_setup* script in place of the ISO's *setup* script—not radically different from a canonical update.

3.3.4.2. Phase 1 for chroot Cells

For chroot cells, the first phase is effected by running the *up_cell* script in the chroot cell as a normal command without needing to reboot anything (TODO really soon).

3.3.4.3. Phase 2 for All Cells (portdb update)

For all cells (but see Section 3.2.2 for some important nuances), the second phase is handled by the *portdb* script and its update lock. Usually this change occurs some time after the new release has been formally announced. If a senior cell (see the same section) for the *release* does not exist in the commonwealth when that occurs, the versioned build config files are inactively held in an available senior cell until an appropriate senior cell for them has been established.

3.4. Introducing Layers: *releases*, *symports*, *mixes*, *builds*, and *deploys*

[Include target info from *pkgaddconf* prologue (6.16) and update there and here]

3.4.1. Symport Collection Sets

Customary local collection trees in `/usr/ports` (used by prime and usrport cells in a prt pkg commonwealth) are arranged as a directory of collection subdirectories with each collection subdirectory containing one port (package directory) for every package in the collection. This is the data organization **ports -u** works with. The `prt-get prtdir` statements were designed to permit the sysadmin to order the collections such that different port versions of the same package would be selected according to site's preferences with the most-preferred collection earliest in the file. To handle anomalies for particular packages in the ordering, a `prtdir` statement identifying one or more specific ports in a certain collection can be placed before those of the collections more preferred than that certain collection that also contain ports for those packages. Thus, the `prt-get` package selection algorithm finds those ports first and selects them.

A symport organization avoids the need for package-specific `prtdir` statements by returning to an organization of collections such that there is only one port for any package within the entire set of collections. This also eliminates the need to get the order of the `prtdir` statements right—the `prtdir` statements effectively define merely the set of collections `prt-get` will work with, even though **prt-get** processes the symport's set of collections no differently from a prime or usrport cell's set.

As the name suggests, symport layouts use symlinks into a prime or usrport cell's `/usr/ports` tree to achieve this magic. In establishing a symport tree, the upper level directory of collections is identical to that of the prime or usrport cell; however, the next level usually contains only symlinks to package directories that were or will soon be actually built using the symport tree. If the sysadmin wants to change the port of packageX from that of collection xyzzy to that of collection plugh, he/she makes the following changes:

```
rm /usr/symports/xyzzy/packageX
ln -s ../../../../ports/plugh/packageX /usr/symports/plugh
```

resulting in `/usr/symports/plugh/packageX`.

Using symport layouts saves some filesystem space and some **ports -u** processing but complicates dependency resolution somewhat as the **prt-cache --test** commands using subcommands **sysup**, **depinst**, or **grpinst** for the symport cell must be run against the entangled prime or usrport cell's `/usr/ports` tree but using the symport cell's port selection preferences. It would be a good idea to

maintain that translation of the `prtdir` statements as part of the symport cell's maintenance policy (remember ***prt-get*** can be told which `prt-get.conf` file to use as a command-line argument and `prtdir` statements can specify absolute paths for the collection directories).

3.4.2. Mixed Cells

Since symport cells link into a particular prime or usrport cell, an entanglement between these cells is recognized. Cells having such entanglements are said to be *mixed* and sets of entangled cells are called *mixes*. A *mix* is defined by a prime or usrport cell followed by the entangled symport cells sorted by collating sequence; e.g.,

```
server3[YY] server1[ZZ] server2[XX_3.3] server4[WW_3.3]
```

where `server3[YY]` is a prime cell and the rest are all symport cells. The `/usr/prtpkg/mixes` trunk holds these single-line mix definition files and PORTSU lock requests by a mixed cell are automatically converted to a lock request for the mix. It should be apparent a symport cell can be mixed with only one prime or usrport cell. Like groups, names of mixes may not have the same format as names of cells.

3.5. Introducing Relationships: *porters*, *builders*, and *deployers*

[Develop explanation and relate maintenance roles: which suppliers serve which consumers.]

3.6. Introducing Batches: *prtpkgbatch* and its **.prtpkg* files

[Explain *prtpkgbatch* processing and *prtpkg* subcommand micromanaging]

3.7. Introducing New Configuration Files: *build.conf* and *deploy.conf*

[Explain what they do and their formats.]

3.8. Introducing `/usr/prtpkg` and Where To Find Everything

To accommodate these new variables and processing realignments, it is necessary to transparently extend the canonical organization of the data manipulated by CRUX software maintenance. This begins with the addition of a tree to hold the commonwealth-wide extra information. The `/usr` tree is an appropriate location for this trunk because this data must be shareable across multiple systems, but not

in `/usr/share` because much of the data will be too volatile for the traditional role of that tree. So it was decided to appropriate `/usr/prtpkg` for this tree.

The root tier holds the following trunks or anchors. Note that `prtpkg` strives to support symlinking as much as possible; indeed, `/usr/prtpkg` itself can be an anchor, if, for example, you want it in a different filesystem from the filesystem containing `/usr` for some reason like not wanting to expose all of `/usr` to other systems. Just remember to ensure dereferences function across multiple platforms as needed (and understand the canonical CRUX software maintenance packages are in general quite a bit less tolerant of symlinks even within a single cell).

3.8.1. `/usr/prtpkg/release`

These inodes anchor the trees for particular versions of any software distribution having ports collections and associated support data compatible with the CRUX maintenance methodology. They are named in the format *distname-version.release*; e.g., `CRUX-3.2`, `CRUX-3.3`.

3.8.2. `/usr/prtpkg/broadcasts`

This inode is (or anchors) a flat file to which requests and notices to be processed by every cell's commonwealth coordination daemon are appended. The daemons receive the requests by piping a ***tail -f*** command for the queue into their ***while read*** loops. If a cell is not active when a request is added then the request is effectively ignored by that cell. See Section 4.2.2.1.1 for more information.

3.8.3. `/usr/prtpkg/builds`

This inode anchors build-definition trees that can be selected for building ports according to a specific combination of options external to the port definition. These can include hardware models/versions, performance objectives, debugging features, linkage attributes—whatever features are desired to be used by one or more packages for one or more systems that can be imposed on package builds outside of the packages proper (yes, systems, not just cells—package files built under `prtpkg` can be installed into non-`prtpkg` CRUX systems). Build definitions do not preclude any package from overriding pieces of the definition. They express the desires of the sysadmin for the package to support that set of features. However, this is in no way a means to describe common package configuration specifications such as the Gentoo distribution's USE facility provides; rather, it is intended to target run-time options of the system's build tools; e.g., compilers and linkers. Like the dependency facility, it is not designed to be comprehensive and infallible, but only

to facilitate organization and methodologies that enable building, for example, production and debugging versions of a port, hopefully without needing to modify the port itself. Two are provided by the prtpkg package: default and debug, but enterprises can add as many as are useful to them, and perhaps such can be shared as ports have been, to the benefit of all.

3.8.4. /usr/prtpkg/cells

This tree anchors a tree for every cell participating in the commonwealth. Cells are identified via strings conforming to the format described in Section 3.2.6.3 and each cell maintains its own as environment variable PRTPKG_CELL. The cell trees anchored in /usr/prtpkg/cells are described in Section 3.8.15.

3.8.5. /usr/prtpkg/groups

Similar to /usr/prtpkg/cells, /usr/prtpkg/groups anchors trees that define groups of cells and/or groups (a group may include cells and groups) as is useful to the enterprise. Note that a group name cannot conform to the cell name format to preclude being misidentified as a cell. Group definitions that contain no trees are supported. It is expected the inodes in the /usr/prtpkg/groups directory are directories containing relative symlinks to trees in *cells*' or *groups*' trunks as is appropriate; e.g.,

```
cellname    → ../../cells/cellname
groupname   → ../groupname
```

Obviously loops of groupnames are unhelpful; e.g.,

```
groups/group1/group2 → ../group2
groups/group2/group3 → ../group3
groups/group3/group1 → ../group1
```

The prtpkg package currently provides no group definitions.

3.8.6. /usr/prtpkg/mixes

Somewhat similar to /usr/prtpkg/groups, /usr/prtpkg/mixes anchors a tree that defines entanglements of symport cells with a prime or usrport cell (see Section 3.4.2 for details). Note that the name of a mix cannot conform to the cell name format defined in Section 3.2.6.3.

3.8.7. /usr/prtpkg/PRTPKG

The PRTPKG tree is a queue of advisory lock bids and grants for update (exclusive) and shared (unmodifiable) access to commonwealth-scope resources not serialized by other locks. Refer to Section 4.1.3.1 for further details.

3.8.8. /usr/prtpkg/PORTDB

The PORTDB tree is a queue of advisory lock bids and grants for update (exclusive) and shared (unmodifiable) access to all senior cells' /etc/ports trees in the commonwealth (see Section 4.1.3.2 for details).

The subordinate tiers of /usr/prtpkg provide the following trees:

3.8.9. /usr/prtpkg/release/builders

3.8.10. /usr/prtpkg/release/deployers

3.8.11. /usr/prtpkg/release/distfiles

3.8.12. /usr/prtpkg/release/packages

3.8.13. /usr/prtpkg/release/work

3.8.14. /usr/prtpkg/release/PORTSU

A PORTSU tree is a queue of advisory lock bids and grants for update (exclusive) and shared (unmodifiable) access to all prime cells' /usr/ports trees subscribed to the release (see Section 4.1.3.3 for details).

3.8.15. /usr/prtpkg/cells/cell

These inodes anchor the trees for the cells associated with this commonwealth. They are named using the *host[rootfs]* format described in Section 3.2.6.3.

3.8.16. `/usr/prtpkg/cells/cell/prtpkg.txt`

This flat file is the chronological activity log for all prtpkg maintenance performed by this cell.

3.8.17. `/usr/prtpkg/cells/cell/notices`

This inode is (or anchors) a flat file to which notices to be processed by this cell's processes are appended. See Section 4.2.2.1.3 for more information.

3.8.18. `/usr/prtpkg/cells/cell/requests`

This inode is (or anchors) a flat file to which requests to be processed by this cell's request handler daemon are appended. The daemon receives the requests by piping a *tail -f* command for the queue into its *while read* loop. If the cell is not active when a request is added and is a chroot cell, its associated boot cell may be requested to launch the chroot cell to begin processing its request queue. If the inactive cell is a boot cell, then the sysadmin may need to get involved to reboot the prtpkg platform with the *ROOTFS* of the needed boot cell. See Section 4.2.2.1.2 for more information.

3.8.19. `/usr/prtpkg/cells/cell/types`

These inodes anchor a flat file containing a single line of three tokens identifying the cell's type attributes in the order shown: boot or chroot, senior or junior, and prime or usrport or symport; e.g., chroot senior usrport.

3.9. Mapping Old Commands Into New Commands

[relocate from *prtpkg h syntax* (6.2) & explain port creation process support]

4. Processing Organization

We start detailing processing organization by looking at how to keep multiple releases and/or parallel processes on the same or different prtpkg platforms from negatively interfering with each other.

4.1. Resource Serialization (Locks)

To prevent maintenance processes from stepping on each other's toes, there is a need for those processes to communicate their serialization requirements and for prtpkg's logic to coordinate those needs, acting like a traffic light (but it is up to those processes to obey the traffic signals to prevent collisions!). This approach is

termed [*advisory locking*](#). Processes requesting a lock will be blocked until the lock becomes available (for some tasks, grants can be blocked for hours). Processes may examine the lock request queues prior to making lock requests to decide, possibly with user input, whether to do something else rather than wait, or to go ahead with what looks to be a short enough wait. The ability to cancel a pending lock request and relinquish ownership of a held lock when a maintenance process is terminating prematurely is also required.

A multi-prtpkg-platform commonwealth is best served by a [*distributed lock management*](#) solution. As a single-prtpkg-platform commonwealth can grow into such a requirement, a single DLM-capable mechanism should provide all prtpkg advisory locking even when no networked filesystems are involved. The prtpkg resource serialization facility uses its own NFS2-style; i.e, stateless, [*lock file*](#) mechanism built upon shared directories that are queues containing flat files that are lock requests, both pending and granted (KISS). This locking facility does not even need to itself be explicitly serialized as distributed filesystem and time-of-day syscalls are sufficiently granular and atomic for its purposes. The filenames begin with subsecond-precision timestamps, so prtpkg platforms do need to have synchronized clocks ([*NTP*](#) works very well for prtpkg purposes).

4.1.1. **Serialization Classes**

Three serialization classes are used to allow maximum parallelism while ensuring data integrity or cohesiveness to those processes that need it.

The *shared* class provides read-only access to the resource while requiring any modification of the resource be prevented while a shared lock is held. Multiple processes may have shared authorization for the same resource concurrently.

The *update* class provides exclusive access to the resource for read-write purposes. Only one process may hold the update lock for a resource at any time and no shared grants can be in effect when an update right is granted. Further, no shared rights for a resource will be granted as long as the update lock is held. All lock requests for a given resource are processed on a FIFO basis, so a pending update request blocks subsequent shared requests for that lock.

The *floating* class is more accurately a state of an update lock in which the lock is currently disassociated from any process (“given up” by the owning process as opposed to “freed up” or “unlocked”) but remains held by the cell to which it was granted. Disassociation occurs when the updating process is compelled to

terminate without completing the update for which it acquired the lock. Ownership of a floating lock can be subsequently “assumed” by another process in the same cell that will attempt to complete the interrupted update. This might even be an interactive shell should the sysadmin need to resolve the situation without the benefit of suitable automation.

4.1.2. Serialization Operations

When a process needs to change its relationship to a serialized resource, it invokes one of these operations. Processes must **trap** all signals that by default result in process termination as well as the EXIT signal in order to disentangle themselves from any lock queues in which they have active requests or grants. Disentangling requires invocation of **lock_cancel**, **lock_unlock**, or **lock_giveup** as is appropriate.

4.1.2.1. lock_obtain

Locks with which a process has no association are requested via this operation that specifies the resource, class, and any parameters the resource type requires. The operation blocks the requesting process as needed until the lock has been granted.

4.1.2.2. lock_assume

When an update lock has been floated because of a failure to update, only this operation can reacquire the lock so the update can be completed. The requester is expected to be assuming the lock to finish that job and so is distinguished from normal update requests that might already be enqueued. As there can be no other grants against a floating lock, the request is immediately granted. However, the request must be made from the same cell in which the lock was given up. An error is recognized if there is no floating lock to assume.

4.1.2.3. lock_freeup

When a process holds an update lock for a serialized resource and has completed the update but needs to continue processing with a shared grant for the resource, it invokes **lock_freeup**. This operation, only available when an update lock is being processed, is used to convert the lock’s class to shared and to continue processing with the shared authority. Any pending shared requests for the lock that can now be granted will be granted before this operation returns.

4.1.2.4. *lock_cancel*

The process' request for a lock that has not yet been granted is removed from the lock queue by this operation, which is normally invoked from a signal handling routine of the process that requested the lock (otherwise it would still be blocked waiting for the lock). If the request has been granted, this operation is treated as a *lock_unlock* operation.

4.1.2.5. *lock_unlock*

This operation is used to drop use of the locked resource and continue processing. Any pending requests for the lock that can now be granted will be granted before this operation returns.

4.1.2.6. *lock_giveup*

This operation, only available when an update lock is being processed, is used to convert the lock's class to floating for premature process termination (or perhaps there is a need to reboot the cell to complete the update). The operation does not terminate the process, it returns so the process may continue managing its termination.

4.1.3. **Serialization Types**

The specific lock types are listed in the order they must be acquired to preclude process deadlocking. Likewise, lock types must be freed up, given up, or unlocked in the reverse order of their acquisition.

4.1.3.1. **Global Serialization: PRTPKG**

There exist prtpkg updates that can touch just about everything prtpkg, so even though such modifications should be infrequent, non-updating users of those resources need assurance the universe will not shift underneath them. This is the *raison d'être* of the PRTPKG lock. Experience may reveal the need to add more granularity at this high impact locking level. On the other hand, if only one lock can be implemented, this is the one. The resources encompassed by the PRTPKG lock are all resources in the commonwealth not covered by the other lock types—it is the catch-all lock of prtpkg serialization.

Since all other prtpkg processing depends on the stability of these foundational resources, all processes must hold at least a shared PRTPKG lock while processing. If such processes are complex or long-running, they should monitor the PRTPKG lock queue and unlock/reacquire their shared PRTPKG lock at the first opportunity to

yield to any waiting update bidder as needed. The more cells that are sharing the PRTPKG lock, the more important such cooperation becomes.

When a PRTPKG update lock request is made, every other prtpkg process in the commonwealth receives a `USR1` signal it interprets as a quiesce directive. Each process must in response gracefully abort what it is doing if it cannot quickly complete what it is doing or must complete what it is doing. Then the process must unlock its shared PRTPKG grant and immediately either terminate or issue a new shared PRTPKG lock request. Examples of PRTPKG updates include a scheduled outage of the network file system for maintenance; manipulating the makeup of the commonwealth such as the sets of defined cells (all `HOST[ROOTFS]` entities), groups, or releases (the set of *releases* in the `/usr/prtpkg` trunk). Also, changing the release of any cell must be performed while holding the PRTPKG update lock (floating the lock across any actual reboots involved).

4.1.3.2. Driver Config Serialization: PORTDB

The *portdb* scripts can update `/etc/ports` contents of multiple senior cells and only knows what will need to be done for each collection while processing each collection. Consequently the PORTDB lock is designed to serialize the group of resources *portdb* maintenance can affect; i.e., the contents of all senior cells' `/etc/ports` trees. Such update serialization also ensures only one *portdb* command can be processing at a time within a commonwealth (but why should there be a need to run different *portdb* commands at the same time, anyway?). Note this lock does not serialize the senior cells, only their `/etc/ports` trees, but shared PORTDB locks acquired by *portsu* processes sufficiently protect them. Also, *prtpkg* processes do not access `/etc/ports` trees, so they can ignore PORTDB locking; however, *prtpkgbatch* scripts; i.e., `.prtpkg` files, can, and care must be taken in creating such to ensure prtpkg locking is properly utilized.

4.1.3.3. Collection Serialization: PORTSU[*|collection]

PORTSU locks protect collection trees from or during (1) *ports -u* processing of prime and `usrport /usr/ports` trees, (2) non-*ports -u* reconfiguration of `symport /usr/ports` trees, and (3) package building. The lock is requested by a specific cell, but if that cell is entangled with any other cells, the lock request is converted from a cell request into a mix request (see Section 3.4.2 for details).

Note: in the rest of this section, understand the use of *mix* to mean *mix or cell* inclusively.

Two scopes of collection serialization at the same level protect collection trees and they are mutually exclusive for any given process:

4.1.3.3.1. All collections: PORTSU_*

This scope protects every collection in the mix of the lock request to be serialized. It is treated as and behaves as a set of PORTSU locks for every collection in the mix—all processed at the same time during a lock operation.

4.1.3.3.2. One collection: PORTSU_ *collectionname*

This scope locks a single collection in the mix of the lock request.

What is serialized specifically is the content of the requesting cell's /usr/ports directory or, if entangled, the entangled prime or usrport cell's /usr/ports directory) for sharing or updates. The granularity of PORTSU locks is for the mix of the requesting cells. The lock queues are placed in the shared cell trunks of the commonwealth; i.e., every cell has a PORTSU lock queue as a /usr/prtpkg/cells/*cell*/PORTSU trunk or anchor. Entangled cells will anchor a lock queue at /usr/prtpkg/mixes/*mix*/PORTSU that is shared by all the entangled cells (instead of a private cell lock queue).

Requests for PORTSU update locks require a shared PORTDB lock be already granted to the requesting process. Similarly, processes requesting a MAKE update lock are required to already be granted the appropriate shared PORTSU lock.

4.1.3.4. Port Serialization: MAKE_ *portname*

The integrity and cohesion of port directories is maintained through a MAKE lock and only one may be requested or held by a single process and its children at a time. Changing the collection of a symport's package requires a MAKE update lock for the port, as does changing the content of a usrport or symport package directory.

The granularity of MAKE locks is for whatever mix or unmixed cell the requesting process runs within. The MAKE lock queues are placed in the shared mix or unmixed cell trunks of the commonwealth; i.e., in every cell, a /usr/prtpkg/cells/*cell*/MAKE trunk or anchor provides its MAKE lock queue.

4.1.3.5. Build Serialization: WORK_ *portname_version_buildname*

These locks protect the package building work directory for active build and post-build work archive processes.

4.1.3.6. Deploy Serialization: `PKG_portname_version_buildname`

These locks protect the package file for active build and deploy processes. Note that *pkgadd* and *pkgrm* maintain their own serialization of the `/var/lib/pkg/db` file and therefore the cell being modified by their processing.

4.2. Inter-process Communication

Processes communicate with each other both synchronously and asynchronously in a commonwealth via shoulder-tapping provided by:

4.2.1. Signal Processing

Prtpkg processes can raise a `signal(7)` for another process in the same cell to handle. This is accomplished via a *kill* command, system call, or some library call. Prtpkg will not depend upon cross-cell signal support within prtpkg platforms (see Section 4.2.2.1.2 for how prtpkg meets this need).

4.2.2. Shared Files

The creation, modification, or removal of an inode in common namespace by one process for notice and/or processing by another process is a methodology for inter-process communication as old as UNIX (actually, older).

4.2.2.1. Broadcast, Request, and Notices Queues

In the interest of KISS, these queue polling facilities are used to eliminate a prtpkg requirement for ssh with its passwords and/or PKI infrastructure (the networked inode permissions of the broadcast, request, and notices queue paths are sufficiently secure for prtpkg's purposes).

4.2.2.1.1. Commonwealth Broadcast Queue

This shared file needs to be polled by every cell's commonwealth coordination daemon to act upon inter-cell notices and requests to groups of cells, possibly to all cells in the commonwealth. Each commonwealth coordination daemon appends requests germane to its cell to the cell's request queue for handling by the cell's request handling daemon and appends germane notices to its notices file.

4.2.2.1.2. Cell Request Queues

Polling of a cell's request queue by the cell's request handling daemon enables cells to respond to processing tasks assigned by other cells within the commonwealth. This mechanism provides the means to indirectly signal processes in other prtpkg platforms as well as other cells within the same prtpkg platform.

4.2.2.1.3. Cell Notices Queues

This shared file acts as a broadcast file at the cell level, receiving appended lines representing notices of significance only within that cell. It is expected that all processes in the cell poll this file as needed. It is not intended to serve as a logging facility.

5. Package Components

The prtpkg package consists of *dash* and *gawk* scripts. Its state is late alpha. See the TODO item in the *prtpkg h* output for more details. This document ([LibreOffice 5.2 Writer source](#)) and the graphic ([LibreOffice 5.2 Draw source](#)) it embeds are licensed under the [standard Creative Commons Attribution-ShareAlike 4.0 International License](#) as will all documentation for prtpkg in the future, while code will be licensed under the [standard GNU Public License Version 2](#) or higher (see http://dlcusa.net/DLC_License.txt for details).

6. Command Information: Help, Prolog, Sample Outputs

The sample reports are edited with cherry-picked lines and column widths have been shortened to get them to fit better in the pages. They are presented to give you a flavor of what prtpkg has to offer. *Some* of the details of existing outputs have been edited for what they should soon look like, mostly the directory/file reorganization, and some of the latest enhancements are not showing in these samples. The point: the prtpkg package is *not* vaporware, but *is* work in progress. Really, everything at the moment should be marked [in transition to data reorg].

6.1. Output: *prtpkg h* [contains TODO items]

prtpkg is the heart of a unified CRUX package maintenance and organization toolset that logs maintenance activity in a prtpkg.txt file (a browsable log that uses tabs as field separators). That data, along with /var/lib/pkg/db, can be processed by the prtpkginfo command to produce txt and log files of installed ports sorted by package and by collection (log files are formatted txt records without tabs and producing column alignment of the data). Another major feature of the toolset is its support for easily identifying how package files were built, e.g., CFLAGS and all other environment variables at build time, build uname -a and other system (and hypervisor) info, package roster,

tool configurations, port tree snapshots; i.e., all package metadata that can be really useful in diagnosing strange package builds and behaviors. Also, the prtptkgbatch command (a driver for all commands, not just prtptkg commands, needed to perform any given package maintenance task as defined in an input .prtptkg file) by default archives compressed build work directory trees using backgrounded nohup subprocesses that execute while subsequent prtptkg subcommands in the batch are processed.

Of necessity, only the subset of canonical CRUX tools that build ports one package per command or install or remove packages one package per command are supported, but with some prohibitions (listed below), the rest can be invoked outside of the prtptkg command to provide machine-readable information that can be easily converted into .prtptkg files.

NOTE: Rather than require changes in the core system maintenance packages, it is thought better to hide the normal ports, prt-get, prt-cache, pkgadd, pkgrm, and pkgmk binaries on a prtptkg-maintained system and install in their places front-ending executables that filter out with appropriate explanation the verboten uses listed below before transferring control to the original binaries via `execve(2)` or the shell built-in `exec` command as appropriate.

NEVER allow these command types to run in a prtptkg-maintained system **UNDER ANY CIRCUMSTANCES:**

```
prt-get without --cache (always use prt-cache except for prt-get cache)
prt-cache depinst without --test
prt-cache grpinst without --test
prt-cache sysup without --test
```

NEVER allow these command types to run in a prtptkg-maintained system **UNLESS** they have been forked within a prtptkg process tree:

```
pkgadd
pkgmk
pkgrm
prt-cache install without --test
prt-cache remove without --test
prt-cache update without --test
ports -u
```

Otherwise, you will have to manually update the prtptkg data files, a likely tedious, error-prone, and potentially impossible undertaking.

This version (of the new version) supports the `pkg{mk,add,rm}`, `override`, and `test` extensions.

TODO:

- * Finish removing dlcisms and cleaning up the code and doc for the first initial release (port or maybe a git repo).
- * Implement **TODO** prtptkg subcommand **TYPE**s (see **SYNTAX** for details), including implementing `build.conf` and `deploy.conf` file handling.
- * Still a lot of development work to be done on implementing overrides.
- * The front-ending ports, prt-get, prt-cache, pkgmk, pkgadd, and pkgrm modules (or scripts) need to be developed and incorporated.
- * Implement prtptkg.conf file of base override definition sets and any other tweaks associated with particular porter and builder systems, special circumstance, etc.--perhaps supporting environment redefinition as part of prtptkg initialization.
- * Add the kernel .config file to the build environment snapshot.
- * Create a script to massage prt-cache sysup|depinst|grpinst --test output into a new .prtptkg file.

For more help, run `prtptkg h all|syntax|global|prt|pkg`

6.2. Output: `prtptkg h syntax` [contains **TODO** items]

SYNTAX: `prtptkg TYPE PACKAGE BATCH TRY [OVERRIDES]`

where:

TYPE of operation is `<|>|B|D|E|I|M|P|R|S|U|a|b|c|d|e|f|h|r|s:`

Note: `-h` and `--help` are the same as a naked `h`.

TYPE	long-name-----	commands-invoked-or-other-description-----	
<	pre-install	[n/a--run the ports's pre-install script]	TODO
>	post-install	[n/a--run the ports's post-install script]	TODO
B	rebuild	pkgmk -f with -i or -u -- invokes pkgadd	
D	demote	[n/a--move files: SSD>HDD>USB>offsite>null]	TODO
E	establish	[n/a--add a CRUX release]	TODO
I	install	prt-cache install -- invokes pkg{mk,add}	
M	migrate	[n/a--upgrade a CRUX release]	TODO
P	promote	[n/a--move files: SSD<HDD<USB<offsite]	TODO

```

R    remove          prt-cache remove      -- invokes pkgrm
S    make_signature  signify                CRUX-3.3-TODO
U    update          prt-cache update      -- invokes pkg{mk,add}
a    add_package     pkgadd                TODO
b    build_package   pkgmk without -i -f -u -d -uf -um (just make) TODO
c    clean_package   pkgmk -c              TODO
d    download        pkgmk -do             TODO
e    extract         pkgmk -eo             TODO
f    make_footprint  pkgmk -uf            TODO
h    help            [specify one or none: all|syntax|global|pkg|prt]
r    remove_package  pkgrm
s    make_md5sum      pkgmk -um            TODO

```

PACKAGE is the name of the CRUX package to be processed. Note: prtptkg does not permit multiple packages to be specified for any type of operation--each package must have its own prtptkg subcommand invocation.

BATCH defines a chronological ordinal number starting with zero with leading zeros identifying the package maintenance task this prt-get invocation is supporting (see prtptkgbatch for further information). The new version of prtptkg uses a 5-digit number with a leading underscore (indicating it's a new version number), while the old version (which had no prtptkgbatch tool) accepts a naked 3-digit number (the underscore will be deprecated when support for the old version, never released into the wild, is dropped, hopefully before the first release).

TRY is a 2-digit chronological ordinal number starting with zero for the attempt being made to complete the maintenance task identified by the BATCH argument, with a leading zero as needed. The old version allowed alphanumeric values; e.g., t1, t23.

OVERRIDES is a string of semi-colon-separated parameters in the form [!]OVERRIDE, where the optional ! causes disabling instead enabling, and OVERRIDE is one of the following as is pertinent to the TYPE of operation requested (most are currently TODO):

```

[global]: _if _im _r= _uf _um redo test
pkgadd:   acd= acf= af  ar= au
pkgmk:    mc  mcd= mcf= mcm md  mdo meo mf  mi
          mif mim min mkw mns mu  mud muf mum
pkgm:     rr=
prt-cache: paa= pca= pcd= pcf= pcp= pcs= pf  pfr pi
          pif pig= pim pins pkw pma= pns pnsd ppos
          ppre pr= pra= pt= puf pum

```

The test override can be used to show how overrides will affect generated prt-cache, pkgmk, pkgadd, and pkgm commands to support developing the desired prtptkg command before committing to its execution.

6.3. Output: *prtptkg h global* [contains TOTO items]

GLOBAL OVERRIDES sorted alphabetically within groups:

```

_if : use this specification to set mif and pif      [TODO]
_im : use this specification to set mim and pim      [TODO]
_r= : use this specification to set ar= and rr=      [TODO]
_uf : use this specification to set muf and puf      [TODO]
_um : use this specification to set mum and pum      [TODO]
redo: force execution of this prtptkg command even if was successfully
      invoked earlier in any attempt to process this batch
test: report what base default flags would be generated and what runnable
      prt-cache, pkgmk, pkgadd, or pkgm command would be produced by
      any overrides in this prtptkg command, but do not execute that
      command or any others that support it (e.g., mkdir), only report
      the commands (or groups of commands) that would be run if test was
      not invoked. While not intended for use in a batch file, this
      override can be applied globally to all batch commands in a
      prtptkgbatch input file via the prtptkgbatch command line test
      argument. Note: This override is completely unrelated to the
      prt-cache --test flag which requires prt-cache be run to perform
      the test.

```

6.4. Output: *prtptkg h prt* [entirely TODO items]

PRT-CACHE OVERRIDES (TYPES I, U, R) sorted alphabetically within groups: -----

Note: prtptkg only permits flags in the aargs, margs, and rargs strings that have no equivalent prt-cache flags in order to simplify prtptkg's flag processing.

```

paa=: Supply "--aargs=" string for any pkgadd invocation    (--aargs="flg[ flg]...")
      pkgadd flags not also supported as prt-cache flags:

```

```

Specify alternative installation root          (-r, --root <path>)
Upgrade package with the same name          (-u, --upgrade)
    Note: prt-cache update specifies --aargs=-u
          automatically, and that cannot be disabled
          (consider running pkgmk without -f or -i
          instead of prt-cache)
Unsupported pkgadd options (not permitted in prtptkg):  unsupported:
    Print help and exit                      (-h, --help)
    Print version and exit                  (-v, --version)
pca=: Append string to this run's prt-cache configuration (--config-append=string)
pcd=: /usr/prtptkg subdirectory housing prt-get.conf      (uses --config=)
pcf=: Supply alternative configuration file to prt-cache  (--config=filespec)
pcp=: Prepend string to this run's prt-cache configuration (--config-prepend=string)
pcs=: Override string in this run's prt-cache configuration (--config-append=string)
pf : Force install by pkgadd                  (-f, -i, --aargs=-f)
pfr : Force rebuild by pkgmk                 (-fr, --margs=-f)
pi : Force install by pkgadd                 (-f, -i, --aargs=-f)
pif : Ignore footprint by pkgmk              (-if, --margs=-if)
pig=: Prevent installation of listed dependencies  (--ignore=pkg[,pkg]...)
pim : Ignore md5sum by pkgmk                 (-im, --margs=-im)
pins: Enable/disable the ppre and ppos overrides  (--pre-install,
--post-install,
--install-scripts)
pkw : Prevent removal of the work directory by pkgmk  (-kw, --margs=-kw)
pma=: Supply "--margs=" string for any pkgmk invocation  (--margs="flg[ flg]...")
    pkgmk flags not also supported as prt-cache flags:
        Remove package and downloaded files          (-c, --clean)
        Do not build, only check md5sum               (-cm, --check-md5sum)
        Download any missing distfiles before building  (-d, --download)
        Note: prt-cache sets --margs=-d automatically,
              and that cannot be disabled (consider -do)
        Only download missing distfiles                (-do, --download-only)
        Only extract distfiles into work directory     (-eo, --extract-only)
        Ignore new files in a footprint mismatch        (-in, --ignore-new)
        Only check for up-to-date                       (-utd, --up-to-date)
    Unsupported pkgmk options (not permitted in prtptkg):  unsupported:
        Search for and build packages recursively      (-r, --recursive)
        Print help and exit                          (-h, --help)
        Print version and exit                      (-v, --version)
pns : Prevent invocation of the strip command by pkgmk  (-ns, --margs=-ns)
pnsd: Don't parse the default configuration file  (--no-std-config)
ppos: Invoke post-install script after pkgadd in prt-cache  (--post-install,
--install-scripts)
ppre: Invoke pre-install script before pkgmk in prt-cache  (--pre-install,
--install-scripts)
pr= : Supply alternative installation root directory  (--install-root=dirspec)
pra=: Supply "--rargs=" string for any pkgmk invocation  (--rargs="flg[ flg]...")
    pkgmk flags not also supported as prt-cache flags:
        Specify alternative installation root directory  (-r, --root <path>)
    Unsupported pkgmk options (not permitted in prtptkg):  unsupported:
        Print help and exit                          (-h, --help)
        Print version and exit                      (-v, --version)
puf : Update footprint by pkgmk                 (-uf, --margs=-uf)
pum : Update md5sum by pkgmk                    (-um, --margs=-um)

Unsupported prt-cache options (not permitted in prtptkg): -----
    Modifies scope of diff, quickdiff, and dependent  (--all)
    Use cache file for this run (always used by prtptkg)  (--cache)
    Modifies output of listinst                       (--depsort)
    Write build output to log (always used by prtptkg)  (--log)
    Show path info for search, dsearch, list, and depends  (--path)
    Modifies output of sysup                          (--nodeps)
    Modifies scope for diff, quickdiff, and sysup       (--prefer-higher, -ph)
    Modifies output of dependent                      (--recursive)
    Use filter, search patterns as regular expression  (--regex)
    Override the 'prefer-higher' option                (--strict-diff, -sd)
    Dry run, don't actually install anything          (--test)
    Modifies output of dependent                      (--tree)
    Verbose and more verbose for search and list       (-v, -vv)

Unsupported prt-cache commands (not permitted in prtptkg): -----
NOTE: depinst, grpinst, and sysup must NEVER be run in a
prtptkg-maintained system UNLESS defanged using the
--test option. Also, install, remove, and update must
NEVER be run outside of prtptkg on a prtptkg-maintained

```

system.						
cache	cat	current	dependent	depends	depinst	deptree
diff	dsearch	dumpconfig	dup	edit	fsearch	grpinst
help	info	isinst	list	listinst	listlocked	listorphans
lock	ls	printf	quickdep	quickdiff	readme	search
sysup	unlock	version				

6.5. Output: *prtptkg h pkg* [entirely TODO items]

PKGMMK OVERRIDES (TYPES B, I, U, m) sorted alphabetically within groups: -----

mc : Remove package and downloaded files	(-c, --clean)
mcdf= : /usr/prtptkg subdirectory housing pkgmmk.conf	(uses --config-file)
mcf= : Use alternative configuration file	(-cf, --config-file <file>)
mcm : Do not build, only check md5sum	(-cm, --check-md5sum)
md : Download any missing distfiles before building	(-d, --download)
mdo : Only download missing distfiles	(-do, --download-only)
meo : Only extract distfiles into work directory	(-eo, --extract-only)
mf : Build package even if it appears to be up to date	(-f, --force)
mi : Build and install package	(-i, --install)
mif : Build package without checking footprint	(-if, --ignore-footprint)
mim : Build package without checking md5sum	(-im, --ignore-md5sum)
min : Ignore new files in a footprint mismatch	(-in, --ignore-new)
mkw : Keep temporary working directory	(-kw, --keep-work)
mns : Do not strip executable binaries or libraries	(-ns, --no-strip)
mu : Build and install package (as upgrade)	(-u, --upgrade)
mud : Only check for up-to-date	(-utd, --up-to-date)
muf : Only update previously built package file's footprint	(-uf, --update-footprint)
mum : Only update .md5sum file in the port directory	(-um, --update-md5sum)

Unsupported pkgmmk options (not permitted in prtptkg): unsupported:

Note: NEVER allow pkgmmk to run outside of prtptkg in a prtptkg-maintained system!

Print help and exit	(-h, --help)
Search for and build packages recursively	(-r, --recursive)
Print version and exit	(-v, --version)

PKGADD OVERRIDES (TYPES B, I, U, a) sorted alphabetically within groups: -----

acd= : /usr/prtptkg subdirectory housing pkgadd.conf	(unsupported by pkgadd)
acf= : the symlink value to use for /etc/pkgadd.conf	(unsupported by pkgadd)
af : force install, overwrite conflicting files	(-f, --force)
ar= : specify alternative installation root	(-r, --root <path>)
au : upgrade package with the same name	(-u, --upgrade)

Unsupported pkgadd options (not permitted in prtptkg): unsupported:

Note: NEVER allow pkgadd to run outside of prtptkg in a prtptkg-maintained system!

print help and exit	(-h, --help)
print version and exit	(-v, --version)

PKGRM OVERRIDES (TYPES B, I, U, R, r) sorted alphabetically within groups: -----

rr= : specify alternative installation root	(-r, --root <path>)
---	---------------------

Unsupported pkgrm options (not permitted in prtptkg): unsupported:

Note: NEVER allow pkgrm to run outside of prtptkg in a prtptkg-maintained system!

print help and exit	(-h, --help)
print version and exit	(-v, --version)

6.6. Prolog: *prtptkgbatch* [contains TODO items]

Process a prtptkg batch as specified by:

Required Arguments:

\$1: the batch_ID of the prtptkg file to process

\$2: the build_ID of the build configuration to run in [TODO]

Optional arguments that can be specified to be passed as overrides into all prtptkg commands invoked by the batch run.

'redo' [TODO]

'test'

Environment Variables:

PRTPKG_SYS the prtptkg platform this command is running on [TODO]

PRTPKG_PORTS the ports_ID of the release of port collections to be used for building [TODO]

6.7. Prolog: *prtptkglog* [in transition to data reorg]

prtptkglog -- output `/usr/prtptkg/$PRTPKG_PORTSU/prtptkg.txt` in log format (no tabs, fields columnized using blanks) piped into the command defined in the arguments or by default into "less -S" (truncate each line at the screen's right boundary).

6.8. Output: *prtptkglog* [in transition to data reorg]

```
00000000-00:00:00UT -rc t package----- version--- coll build- batch try-
working_directory----- command-----
20170130-14:15:34UT 0 U xorg-libxi 1.7.9-1 xorg pkgsb3 00031 06
/var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f xorg-libxi
20170130-14:22:33UT 0 U mesa3d 12.0.6-1 xorg pkgsb3 00031 06
/var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f mesa3d
20170130-14:22:51UT 0 U libva 1.7.3-1 opt pkgsb3 00031 06
/var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f libva
20170130-14:23:10UT 0 U freeglut 3.0.0-1 opt pkgsb3 00031 06
/var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f freeglut
20170130-14:27:20UT 0 U fontforge 20161012-1 opt pkgsb3 00031 06
/var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f fontforge
20170130-15:07:58UT 0 U firefox 51.0-1 opt pkgsb3 00031 06
/var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f firefox
```

6.9. Output: *cat /usr/prtptkg/cells/dlcz[ZD]/CRUX-3.2/pkgsb3/log/00031/06.log*

```
==== prtptkgbatch is running batch 00031 try 06 in root [ZD] of host dlcz at Mon Jan 30 13:32:31 UTC 2017
-rw-r--r-- 1 root root 1.9K Jan 30 13:31 /usr/prtptkg/systems/dlcz[ZD]/CRUX-3.2/00031.prtptkg
# sysup 2017-01-29
## do_prtptkg U openssl
## do_prtptkg U libpcre
## do_prtptkg U flex
## do_prtptkg U libspiro
## do_prtptkg U kbd
## do_prtptkg U libyaml
## do_prtptkg U orc
## do_prtptkg U curl
## do_prtptkg U elfutils
## do_prtptkg U speex
## do_prtptkg U sed
## do_prtptkg U openldap
## do_prtptkg U nss
## do_prtptkg U nfs-utils
## do_prtptkg U btrfs-progs
## do_prtptkg U krb5
## do_prtptkg U glib
#
# The first try to U python3-setuptools failed for the situation described
# in the port's new README, so attempt the solution it documents...
#
## do_prtptkg I python3-pip
#
# The second try failed looking for module six -- since package six is
# installed, try installing python3-six (all dependencies already installed)...
#
## do_prtptkg I python3-six
#
# The 3rd try complains about module packaging not found--try installing all
# 6c37 python3-* ports with their uninstalled deps...
#
## do_prtptkg I log4cplus
## do_prtptkg I leptonica
## do_prtptkg I libwebp
#
# Alan's tesseract fails checksum, so select 6c37's by adding it to the 1st
# 6c37 prtdir... Only multiple ports per prtdir is breaking prtlist somehow.
# Worry about that latter, instead create a _symlinks collection that holds
# symlinks to all the ports previously in "prtdir collection:port[, port]..."
# statements, which made prtlist happy...
#
# Only 6c37's tesseract won't build, either--meanwhile, there is a new
# pre-install script for python3-setuptools that needs to be run for it to
# install properly, so modified prtptkg to default to --install-scripts for
# updates, and get rid of the other unnecessary 6c37 ports deps...
#
do_prtptkg R libwebp
do_prtptkg R leptonica
do_prtptkg R log4cplus
do_prtptkg U python3-setuptools
do_prtptkg U llvm
do_prtptkg U taglib
do_prtptkg U gobject-introspection
do_prtptkg U mako
do_prtptkg U libvirt
do_prtptkg U xorg-libxi
do_prtptkg U mesa3d
do_prtptkg U libva
do_prtptkg U freeglut
do_prtptkg U fontforge
do_prtptkg U firefox
==== prtptkgbatch is now running the above commands:
Mon Jan 30 13:32:31 UTC 2017 -- whatprt libwebp => /usr/ports/contrib 0.5.1-1
```

```

Mon Jan 30 13:32:31 UTC 2017 -- whatpkg libwebp => contrib      0.5.1-1      pkgusb3      00031      04
Mon Jan 30 13:32:31 UTC 2017 -- Invoking /com/bin/prtpkg R libwebp _00031 06
Mon Jan 30 13:32:32 UTC 2017 -- Invocation succeeded
==== prtpkgbatch command <do_prtpkg R libwebp> returned 0
Mon Jan 30 13:32:32 UTC 2017 -- whatprt leptonica => /usr/ports/6c37 1.74.1-1
Mon Jan 30 13:32:32 UTC 2017 -- whatpkg leptonica => 6c37      1.74.1-1      pkgusb3      00031      04
Mon Jan 30 13:32:32 UTC 2017 -- Invoking /com/bin/prtpkg R leptonica _00031 06
Mon Jan 30 13:32:33 UTC 2017 -- Invocation succeeded
==== prtpkgbatch command <do_prtpkg R leptonica> returned 0
Mon Jan 30 13:32:33 UTC 2017 -- whatprt log4cplus => /usr/ports/6c37 1.2.0-1
Mon Jan 30 13:32:33 UTC 2017 -- whatpkg log4cplus => 6c37      1.2.0-1      pkgusb3      00031      04
Mon Jan 30 13:32:33 UTC 2017 -- Invoking /com/bin/prtpkg R log4cplus _00031 06
Mon Jan 30 13:32:34 UTC 2017 -- Invocation succeeded
==== prtpkgbatch command <do_prtpkg R log4cplus> returned 0
==== prtpkgbatch archived port _symlinks/python3-setuptools
====
in /usr/prtpkg/systems/dlcz[20]/CRUX-3.2/log/_/00031/06/python3-setuptools.port.tar.bz2

```

[...]

```

==== prtpkgbatch while read EOF encountered
==== prtpkgbatch finished running the above commands with status 0
==== Running prtpkginfo -sup -dbg -lpf -lcf -bld -bat -try -dat -typ
The pkg/db and/or prtpkg.log files are more recent than any
prtpkg_by_pkg.*.txt file in the working directory. Thus,
the -upd option is in effect even if it was not specified
Generating /var/lib/pkg/prtpkg_by_pkg.20170130-150758.txt...
Generating /var/lib/pkg/prtpkg_by_col.20170130-150758.txt...
Generating /var/lib/pkg/prtpkg_by_pkg.20170130-150758.log from /var/lib/pkg/prtpkg_by_pkg.20170130-150758.txt...
Generating /var/lib/pkg/prtpkg_by_col.20170130-150758.log from /var/lib/pkg/prtpkg_by_col.20170130-150758.txt...
*** prtpkg_by_pkg.20170130-055017.txt      2017-01-30 05:50:17.471873661 +0000
--- prtpkg_by_pkg.20170130-150758.txt      2017-01-30 15:07:58.752057202 +0000
*****
*** 86 ***
! firefox      50.1.0-1      [same]      opt      pkgusb3      00025      18      20170113-
17:59:55UT      B      /usr/ports/opt/firefox      pkgmk -d -f -u -if -kw -ns
--- 86 ----
! firefox      51.0-1      [same]      opt      pkgusb3      00031      06      20170130-
15:07:58UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f firefox
*****
*** 92 ***
! fontforge      20150824-1      [same]      opt      pkgusb3      00025      17      20170113-
17:12:51UT      B      /usr/ports/opt/fontforge      pkgmk -d -f -u -if -kw -ns
--- 92 ----
! fontforge      20161012-1      [same]      opt      pkgusb3      00031      06      20170130-
14:27:20UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f fontforge
*****
*** 94 ***
! freeglut      2.8.1-1      [same]      opt      pkgusb3      00025      13      20170113-
16:36:47UT      B      /usr/ports/opt/freetglut      pkgmk -d -f -u -if -kw -ns
--- 94 ----
! freeglut      3.0.0-1      [same]      opt      pkgusb3      00031      06      20170130-
14:23:10UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f freeglut
*****
*** 127 ***
! gobject-introspection      1.48.0-1      [same]      opt      pkgusb3      00025      11
-kw -ns      20170113-11:29:45UT      B      /usr/ports/opt/gobject-introspection      pkgmk -d -f -u -if
--- 127 ----
! gobject-introspection      1.50.0-1      [same]      opt      pkgusb3      00031      06
20170130-14:12:47UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns
--install-scripts -f gobject-introspection
*****
*** 187 ***
! leptonica      1.74.1-1      [same]      6c37      pkgusb3      00031      04      20170130-
05:09:46UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f leptonica
--- 187 ----
! leptonica      [not installed] 1.74.1-1      6c37      pkgusb3      00031      04      20170130-
05:09:46UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f leptonica
*****
*** 268 ***
! libva      1.7.2-1      [same]      opt      pkgusb3      00025      12      20170113-
12:21:29UT      B      /usr/ports/opt/libva      pkgmk -d -f -u -if -kw -ns
--- 268 ----
! libva      1.7.3-1      [same]      opt      pkgusb3      00031      06      20170130-
14:22:51UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f libva
*****
*** 271 ***
! libvirt      2.5.0-2      [same]      nullspoon      pkgusb3      00025      11      20170113-
11:29:10UT      B      /usr/ports/nullspoon/libvirt      pkgmk -d -f -u -if -kw -ns
--- 271 ----
! libvirt      3.0.0-1      [same]      nullspoon      pkgusb3      00031      06      20170130-
14:15:18UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f libvirt
*****
*** 277 ***
! libwebp      0.5.1-1      [same]      contrib      pkgusb3      00031      04      20170130-
05:10:09UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f libwebp
--- 277 ----
! libwebp      [not installed] 0.5.1-1      contrib      pkgusb3      00031      04      20170130-
05:10:09UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f libwebp
*****
*** 291,292 ***
! llvm      3.9.0-2      [same]      opt      pkgusb3      00025      07      20170113-
07:48:21UT      B      /usr/ports/opt/llvm      pkgmk -d -f -u -if -kw -ns
! log4cplus      1.2.0-1      [same]      6c37      pkgusb3      00031      04      20170130-
05:08:48UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f log4cplus
--- 291,292 ----
! llvm      3.9.1-3      [same]      opt      pkgusb3      00031      06      20170130-
14:11:42UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f llvm

```

```

! log4cplus      [not installed] 1.2.0-1      6c37      pkgsb3      00031      04      20170130-
05:08:48UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f log4cplus
*****
*** 303 ****
! mako           1.0.1-1      [same]      opt      pkgsb3      00025      11      20170113-
11:33:45UT      B      /usr/ports/opt/mako      pkgmk -d -f -u -if -kw -ns
--- 303 ----
! mako           1.0.4-1      [same]      opt      pkgsb3      00031      06      20170130-
14:12:53UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f mako
*****
*** 307 ****
! mesa3d         12.0.5-1      [same]      xorg      pkgsb3      00025      12      20170113-
12:15:30UT      B      /usr/ports/xorg/ mesa3d      pkgmk -d -f -u -if -kw -ns
--- 307 ----
! mesa3d         12.0.6-1      [same]      xorg      pkgsb3      00031      06      20170130-
14:22:33UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f mesa3d
*****
*** 391 ****
! python3-setuptools 32.3.1-1      [same]      6c37      pkgsb3      00025      10
20170113-11:11:40UT      I      /var/log/pkgbuild/_/00025/10      prt-cache install --margs="-if"
-kw -ns --install-scripts --aargs="-f"      python3-setuptools
--- 391 ----
! python3-setuptools 34.1.0-1      [same]      _symlinks      pkgsb3      00031      06
20170130-13:32:44UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns
--install-scripts -f python3-setuptools
*****
*** 452 ****
! taglib         1.11-1      [same]      opt      pkgsb3      00025      07      20170113-
06:56:31UT      B      /usr/ports/opt/taglib      pkgmk -d -f -u -if -kw -ns
--- 452 ----
! taglib         1.11.1-1      [same]      opt      pkgsb3      00031      06      20170130-
14:12:04UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f taglib
*****
*** 591 ****
! xorg-libxi     1.7.8-1      [same]      xorg      pkgsb3      00025      12      20170113-
12:00:43UT      B      /usr/ports/xorg/xorg-libxi      pkgmk -d -f -u -if -kw -ns
--- 591 ----
! xorg-libxi     1.7.9-1      [same]      xorg      pkgsb3      00031      06      20170130-
14:15:34UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f xorg-libxi
*** prtpkg_by_col.20170130-055017.txt      2017-01-30 05:50:17.474873675 +0000
--- prtpkg_by_col.20170130-150758.txt      2017-01-30 15:07:58.755057215 +0000
*****
*** 23,24 ****
! leptonica      1.74.1-1      [same]      6c37      pkgsb3      00031      04      20170130-
05:09:46UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f leptonica
! log4cplus      1.2.0-1      [same]      6c37      pkgsb3      00031      04      20170130-
05:08:48UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f log4cplus
--- 23,24 ----
! leptonica      [not installed] 1.74.1-1      6c37      pkgsb3      00031      04      20170130-
05:09:46UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f leptonica
! log4cplus      [not installed] 1.2.0-1      6c37      pkgsb3      00031      04      20170130-
05:08:48UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f log4cplus
*****
*** 27 ****
- python3-setuptools 32.3.1-1      [same]      6c37      pkgsb3      00025      10
20170113-11:11:40UT      I      /var/log/pkgbuild/_/00025/10      prt-cache install --margs="-if"
-kw -ns --install-scripts --aargs="-f"      python3-setuptools
--- 26 ----
*****
*** 29 ****
--- 29 ----
+ python3-setuptools 34.1.0-1      [same]      _symlinks      pkgsb3      00031      06
20170130-13:32:44UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns
--install-scripts -f python3-setuptools
*****
*** 86 ****
! libwebp        0.5.1-1      [same]      contrib      pkgsb3      00031      04      20170130-
05:10:09UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f libwebp
--- 86 ----
! libwebp        [not installed] 0.5.1-1      contrib      pkgsb3      00031      04      20170130-
05:10:09UT      I      /var/log/pkgbuild/_/00031/04      prt-cache install -if -kw -ns --install-scripts -f libwebp
*****
*** 258 ****
! libvirt        2.5.0-2      [same]      nullspoon      pkgsb3      00025      11      20170113-
11:29:10UT      B      /usr/ports/nullspoon/libvirt      pkgmk -d -f -u -if -kw -ns
--- 258 ----
! libvirt        3.0.0-1      [same]      nullspoon      pkgsb3      00031      06      20170130-
14:15:18UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f libvirt
*****
*** 299 ****
! firefox        50.1.0-1      [same]      opt      pkgsb3      00025      18      20170113-
17:59:55UT      B      /usr/ports/opt/firefox      pkgmk -d -f -u -if -kw -ns
--- 299 ----
! firefox        51.0-1      [same]      opt      pkgsb3      00031      06      20170130-
15:07:58UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f firefox
*****
*** 302,303 ****
! fontforge      20150824-1      [same]      opt      pkgsb3      00025      17      20170113-
17:12:51UT      B      /usr/ports/opt/fontforge      pkgmk -d -f -u -if -kw -ns
! freeglut       2.8.1-1      [same]      opt      pkgsb3      00025      13      20170113-
16:36:47UT      B      /usr/ports/opt/freetglut      pkgmk -d -f -u -if -kw -ns
--- 302,303 ----
! fontforge      20161012-1      [same]      opt      pkgsb3      00031      06      20170130-
14:27:20UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f fontforge
! freeglut       3.0.0-1      [same]      opt      pkgsb3      00031      06      20170130-
14:23:10UT      U      /var/log/pkgbuild/_/00031/06      prt-cache update -if -kw -fr -ns --install-scripts -f freeglut
*****
*** 318 ****

```



```

! gobject-introspection 1.48.0-1 [same] opt pkgsb3 00025 11
20170113-11:29:45UT B /usr/ports/opt/gobject-introspection pkgmk -d -f -u -if
-kw -ns
--- 318 ---
! gobject-introspection 1.50.0-1 [same] opt pkgsb3 00031 06
20170130-14:12:47UT U /var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns
--install-scripts -f gobject-introspection
*****
*** 379 ***
! libva 1.7.2-1 [same] opt pkgsb3 00025 12 20170113-
12:21:29UT B /usr/ports/opt/libva pkgmk -d -f -u -if -kw -ns
--- 379 ---
! libva 1.7.3-1 [same] opt pkgsb3 00031 06 20170130-
14:22:51UT U /var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f libva
*****
*** 389 ***
! llvm 3.9.0-2 [same] opt pkgsb3 00025 07 20170113-
07:48:21UT B /usr/ports/opt/llvm pkgmk -d -f -u -if -kw -ns
--- 389 ---
! llvm 3.9.1-3 [same] opt pkgsb3 00031 06 20170130-
14:11:42UT U /var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f llvm
*****
*** 393 ***
! mako 1.0.1-1 [same] opt pkgsb3 00025 11 20170113-
11:33:45UT B /usr/ports/opt/mako pkgmk -d -f -u -if -kw -ns
--- 393 ---
! mako 1.0.4-1 [same] opt pkgsb3 00031 06 20170130-
14:12:53UT U /var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f mako
*****
*** 454 ***
! taglib 1.11-1 [same] opt pkgsb3 00025 07 20170113-
06:56:31UT B /usr/ports/opt/taglib pkgmk -d -f -u -if -kw -ns
--- 454 ---
! taglib 1.11.1-1 [same] opt pkgsb3 00031 06 20170130-
14:12:04UT U /var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f taglib
*****
*** 519 ***
! mesa3d 12.0.5-1 [same] xorg pkgsb3 00025 12 20170113-
12:15:30UT B /usr/ports/xorg/mesa3d pkgmk -d -f -u -if -kw -ns
--- 519 ---
! mesa3d 12.0.6-1 [same] xorg pkgsb3 00031 06 20170130-
14:22:33UT U /var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f mesa3d
*****
*** 601 ***
! xorg-libxi 1.7.8-1 [same] xorg pkgsb3 00025 12 20170113-
12:00:43UT B /usr/ports/xorg/xorg-libxi pkgmk -d -f -u -if -kw -ns
--- 601 ---
! xorg-libxi 1.7.9-1 [same] xorg pkgsb3 00031 06 20170130-
14:15:34UT U /var/log/pkgbuild/_/00031/06 prt-cache update -if -kw -fr -ns --install-scripts -f xorg-libxi
==== Results of revdep:
libreoffice
opera
syslinux
tesseract
==== End of revdep results.
==== Do not forget to run rejmerge

```

6.10. Output: `cat /usr/prtpkg/CRUX-3.2/prtpkg_by_col.20170130-161126.log`

```

package----- pkg/db_version- port_version collection-- build- batch try date_time----- type
firefox-java-plugin 1.7.0-2 !? !? !? !? !? !?
gc 7.4.2-1 !? !? !? !? !? !?
leptonica [not installed] 1.74.1-1 6c37 pkgsb3 00031 04 20170130-05:09:46UT I
log4cplus [not installed] 1.2.0-1 6c37 pkgsb3 00031 04 20170130-05:08:48UT I
python3-pip 9.0.1-1 [same] 6c37 pkgsb3 00031 01 20170129-18:05:33UT I
python3-pyflakes 1.5.0-1 [same] 6c37 pkgsb3 00031 05 20170130-05:50:10UT I
rubberband 1.8.1-1 [same] 6c37 pkgsb3 00025 11 20170113-11:27:17UT B
vamp-plugin-sdk 2.6-2 [same] 6c37 pkgsb3 00025 07 20170113-06:55:13UT B
python3-setuptools 34.1.0-1 [same] _symlinks pkgsb3 00031 06 20170130-13:32:44UT U
tesseract 3.04.01-1 [same] _symlinks pkgsb3 00031 05 20170130-05:49:59UT I
atkmm 2.24.2-1 [same] contrib pkgsb3 00025 18 20170113-20:52:49UT B
boost 1.63.0-1 [same] contrib pkgsb3 00025 07 20170113-07:10:43UT B
bridge-utils 1.5-1 [same] contrib pkgsb3 00025 01 20170109-01:48:19UT B
cairomm 1.12.0-1 [same] contrib pkgsb3 00025 12 20170113-12:07:19UT B
denyhost 2.9-1 [same] contrib pkgsb3 00025 07 20170113-07:49:25UT B
dev86 0.16.21-1 [same] contrib pkgsb3 00025 01 20170109-01:50:43UT B
dmidecode 2.12-1 [same] contrib pkgsb3 00025 01 20170109-01:51:11UT B
docbook-xml [not installed] 4.5-6 contrib pkgsb3 00012 04 20161225-17:08:13UT I

```

6.11. Output: `pkg_basenames` [in transition to data reorg]

```

BBS xfwm4-themes(4.10.0-1) D '/usr/share/themes/BBS'
BC.3x.gz ncurses(6.0-3) F '/usr/share/man/man3/BC.3x.gz'
' -> 'curs_termcap.3x.gz'
' => '/usr/share/man/man3/curs_termcap.3x.gz'
BCC.pm F '/usr/lib/perl5/5.22/ExtUtils/CBuilder/Platform/Windows/BCC.pm'
BDCE.h F '/usr/include/llvm/Transforms/Scalar/BDCE.h'
BER.h F '/usr/include/c++/5.4.0/gnu/java/security/ber/BER.h'
BEREncodingException.h F '/usr/include/c++/5.4.0/gnu/java/security/ber/BEREncodingException.h'
BERReader.h F '/usr/include/c++/5.4.0/gnu/java/security/ber/BERReader.h'
BERValue.h F '/usr/include/c++/5.4.0/gnu/java/security/ber/BERValue.h'
BF_cbc_encrypt.3ssl.gz openssl(1.0.2k-1) F '/usr/share/man/man3/BF_cbc_encrypt.3ssl.gz'
' -> 'blowfish.3ssl.gz'
' => '/usr/share/man/man3/blowfish.3ssl.gz'
BF_cfb64_encrypt.3ssl.gz openssl(1.0.2k-1) F '/usr/share/man/man3/BF_cfb64_encrypt.3ssl.gz'
' -> 'blowfish.3ssl.gz'
' => '/usr/share/man/man3/blowfish.3ssl.gz'

```

6.12. Prolog: *localize_ports* [in transition to data reorg]

If no args, configures all ports after the initial portsu script for maintenance on this system using the enhanced framework; otherwise, configures a new port for the enhanced framework (\$1=collection \$2=package) and is meant to be called from the prtptkg script.

6.13. Prolog: *missing_packages* [in transition to data reorg]

Show packages depended on that do not exist, sorted by:

```

name    of collection referencing missing package
name    of package   referencing missing package
version of package   referencing missing package
name    of missing depended on package

```

The only argument is the target-ID of the running system to confirm this invocation is not merely a help request.

6.14. Prolog: *missing_packages_doit* (gawk) [in transition to data reorg]

from /usr/CRUX/prtlist.*.txt, write missing_ports records to stdout (expected to be redirected to /tmp/missing_ports.pkgs. The /tmp/missing_packages.pkgs file created by the missing_packages script is read into an array to provide the definitive list of all known packages.

6.15. Output: *misspkglog* [in transition to data reorg]

6c37	rubberband	1.8.1-1	vamp-plugins-sdk
6c37-git	connman	git-1	gnutils
6c37-git	nctelegram	git-1	pytg3
6c37-git	od6config	git-1	fglrx
6c37-git	urwid3	1.3.1-1	setuptools3
alan	anki	2.0.39-1	distribute
alan	docbook2x	0.8.8-1	docbook
alan	dovecot	2.2.25-6	tcp_wrappers
alan	freevo	1.9.0-1	pygame
alan	gocr	0.50-1	netpbm
alan	hplip	3.11.10-1	foomatic-filters
alan	k2pdfopt	2.32-1	netpbm
alan	kodi	16.1-Jarvis-6	freetype2
alan	lirc-xmms-plugin	1.4-1	xmms
alan	nfs-utils	1.3.3-4	udev
alan	p5-calendar-japanese-holiday	0.03-1	p5-calendar
alan	p5-cam-pdf	1.60-1	p5-crypt-rc4
alan	p5-dbd-sqlite	1.46-1	sqlite
alan	p5-module-signature	0.70-1	p5-digest-sha
alan	p5-net-dbus	1.0.0-1	p5-xml-twig
alan	p5-text-pdf	0.29a-1	p5-crypt-rc4
alan	p5-xml-xql	0.68-1	p5-date-manip
alan	p5-xml-xql	0.68-1	p5-xml
alan	p5-yaml	1.15-2	p5-spiffy
alan	p5-yaml	1.15-2	p5-test-base
alan	py-send2trash	1.3.0-1	distribute
alan	pyqt	4.9.1-1	qt
alan	pysqlite	2.4.1-1	sqlite
alan	spamassassin	3.4.1-2	p5-lwp
alan	timidity-sgm	2.01-1	timidity

6.16. Prolog: *pkgaddconf* [targets contain TODO items]

ensure the /etc/pkgadd.conf symlink points to the version defined by the specified target argument; e.g., for argument xyzzy[plugh], the symlink should be or become

```
/etc/pkgadd.conf -> /usr/CRUX/xyzzy[plugh]/pkgadd.conf
```

If successful, the script writes the target component of the previous symlink (may be the same) to stdout.

-or- omit an argument to output the target component of the current symlink to stdout (this does not require the caller to be the superuser).

return: Status code 0 is returned to indicate the data written to stdout is the expected value; otherwise, an error message is written to stdout and a non-zero status code is returned.

targets: define a CRUX port/package building structure to support a particular set of port/package maintenance activities. Targets come in three types having one of the following formats as appropriate:
 release: CRUX-\$V.\$R; e.g., CRUX-3.2

system: \$HOST[\$ROOT]; e.g., mydevbox[SSD3], where:
 \$HOST is the output of the hostname command for the targeted system
 \$ROOT is the enterprise's identifier for the targeted root filesystem of that system, ideally a volume, partition, or filesystem label
 shared: an enterprise-meaningful string without any brackets; e.g., shared, common, production, testing, marketing

A release target is used for ports -u processing by one CRUX software maintenance system designated to do so on behalf of all other CRUX systems in the enterprise running the same version of CRUX. It defines the standard /usr/ports tree of prtdirs on that system.

A system target is used for building and installing/removing packages for the targeted system (which is not necessarily the system performing the port/package maintenance). A system target's prt-get.config file defines the /usr/prtptkg/\$system/ports tree of prtdirs that deploy symlinks to redirect to the release ports directories that the system uses (see the validate_system command for more information). For example,
 /usr/prtptkg/systems/xyzy[plugh]/ports/core/gcc -> /usr/ports.core/gcc

A shared target is exactly like a system target except it identifies a grouping of systems sharing the same building configuration. One system should be designated to perform the actual port/package maintenance on the behalf of the group--it need not even be in that group.

6.17. Prolog: *prtlist* [in transition to data reorg]

For all packages defined within /etc/prt-get.conf (via prtmdir statements), produce a sorted list of the packages, with different ports of the same name sorted in prtmdir access order (the first is the port that prt-get will act upon, and the others will have a single double quotation mark appended; i.e., a "ditto" indicator). Each port line includes the prtmdir path, the version-release tuple, and an optional indicator that the package does not contain a "Depends on:" record. Each port line is followed by lines containing the Depends on: values, indented by two spaces, and listed in the order they are specified in the "Depends on:" record.

This script takes no parameters. It should be run after a "ports -u" command (which is one reason you should instead use the portsu script on this system) or after the order and/or content of the prtmdir statements in /etc/prt-get.conf is modified.

The output is put into a file named as /usr/ports/prtlist.YYYYMMDD-HHMMSS.txt and if an earlier file exists, the two are compared and the result is placed into /usr/ports/prtlist.diffs, then the older file is compressed using bzip2 and moved into the /usr/ports/prtlist_past directory.

A /tmp/prtlist.\$\$ debug file contains details of processing helpful in diagnosing any misbehavior.

6.18. Prolog: *prtlist_packages* (gawk) [in transition to data reorg]

from /usr/prtptkg/\$PRTPKG_PORTS/prtlist.*.txt, extract list of available packages

6.19. Output: *prtlist* [in transition to data reorg]

babl	/usr/ports/opt	0.1.16-1	
babl"	/usr/ports/teatime	0.1.18-1	[no Depends On]
babl"	/usr/ports/deepthought	0.1.18-1	
backlight	/usr/ports/6c37-git	git-1	
git			
baloo	/usr/ports/kde4	4.14.3-1	
kdepimlibs			
xapian			
kfilemetadata			
baloo"	/usr/ports/kf5	5.30.0-1	
gtk			
kfilemetadata			
kidletime			
kio			
lmdb			
gtk			
baloo-widgets	/usr/ports/kf5	16.12.1-1	
kdelibs4support			
baloo			
bash	/usr/ports/core	4.3.48-1	
ncurses			
readline			
bash-completion	/usr/ports/opt	2.1-2	

bash			
bash-completion-extras	/usr/ports/deepthought	0.0-1	
bash-completion			
bash-git-prompt	/usr/ports/deepthought	1.2-1	[no Depends On]
bashish	/usr/ports/romster	2.2.4-1	
bashish"	/usr/ports/6c37	2.2.4-1	

6.20. Prolog: *prt pkg_symlink* (gawk)

from piped input for an `"/bin/ls -lh"` command for a single symlink, extract the redirection string and write it to stdout

6.21. Prolog: *prt pkginfo* [in transition to data reorg]

prt pkginfo -- see catted help below for documentation

input *prt pkg*.txt package maintenance record format (tab field separators):
 \$1 [YYYYMMDDhhmmss] \$2 \$3 \$4 \$5 \$6 \$7 \$8 \$9 \$10 \$11
 00000000-00:00:00UT rc type pkg version collection build batch try pwd cmd

output txt record format (tab field separators) and associated log columns:
 \$1 \$2 \$3 \$4 \$5 \$6 \$7 \$8 \$9 \$10 \$11
 pkg_name pkg_ver out_ver prt_col prt_bld prt_bat prt_try prt_dat prt_typ prt_pwd prt_cmd
 always always always always -bld -bat -try -dat -typ -pwd -cmd

6.22. Output: *prt pkginfo -h* [in transition to data reorg]

prt pkginfo: Create *prt pkg_by_{pkg,col}.ststmp.{txt,log}* files from `/usr/prt pkg/systems/$system/var.lib.pkg/db` and `/usr/prt pkg/system/$system/prt pkg.txt` files. The txt files are created if the input files have been modified since the most recent *prt pkg_by_pkg*.txt* file in the working directory was created, if any (or this can be explicitly requested via the `-upd` flag). The log files are created from the txt files if requested. If there are current *prt pkg_by_{pkg,col}.txt* files in the working directory, *diffs* files against any new .txt files are created and the older .txt files are compressed and moved into the working directory's *prt pkg_past* subdirectory that will be created if necessary.

Note: Unknown flags are silently ignored.

Processing option flags:

- sup: change the working directory to `/usr/prt pkg/$PRTPKG_PORTS` (requires superuser authority).
 - upd: requests update of the *prt pkg*.txt* files in the working directory even if that does not seem to be necessary.
 - lpf: requests output of the *by_pkg* log as the file *prt pkg_by_pkg.log* file in the working directory.
 - lcf: requests output of the *by_col* log as the file *prt pkg_by_col.log* file in the working directory.
 - lps: requests output of the *by_pkg* log to stdout.
 - lcs: requests output of the *by_col* log to stdout.
 - dbg: log debug messages to `/tmp/prt pkg.$USER.debug`
- Note: -lpf and -lps are two sides of a coin--the file option prevails if both are specified; if neither is specified, the file/stream is not output.
 The same is true of the -lcf and -lcs options.

Optional log file fields selected by flags:

- bld: include build ID in log files|streams
- bat: include batch ID in log files|streams
- try: include try ID in log files|streams
- dat: include date in log files|streams
- typ: include *prt pkg* record type in log files|streams
- pwd: include cmd's working directory in log files|streams
- cmd: include low-level command in log files|streams
- all: include all of the preceding fields in log files|streams

6.23. Prolog: *validate_builds* [very early new program]

Ensure the target system's tree of ports used in the builds version of `/etc/prt-get.conf` (referred to as the builds ports tree) matches the packages currently installed (or uninstalled but built) as recorded in `/usr/prt pkg/$PRTPKG_PORTS/prt pkg.txt`. If multiple ports for an uninstalled package are available, the one chosen is the first in the *prtdir* order according to

the portsu version of prt-get.conf, which uses the _symlinks pseudo-collection to contain symlinks to any cherry-picked ports; e.g.,
 /usr/ports/_symlinks/xyzy -> /usr/ports/plugh/xyzy
 Inodes in builds ports collection subdirectories are symlinks to the portsu ports tree; e.g.,
 /usr/CRUX/\$hrid/ports/plugh/xyzy -> /usr/ports/plugh/xyzy
 Installed ports that have no port history will be presumed to be associated with the portsu config's normal selection (the highest prtdir having a port for that package).
 This script takes only optional '-h' and '--help' arguments.

6.24. Prolog: *validate_symports* [in transition from varports]

Ensure the symports tree used in the builds version of /etc/prt-get.conf matches the packages either currently installed or uninstalled but built as recorded in /usr/prtptkg/\$PRTPKG_PORTS/prtptkg.txt. If multiple ports for an uninstalled package are available, the one chosen is the first in the prtdir order according to the portsu version of prt-get.conf, which uses the _symlinks pseudo-collection to contain symlinks to any cherry-picked ports; e.g.,
 /usr/ports/_symlinks/xyzy -> /usr/ports/plugh/xyzy
 Inodes in symports collection subdirectories are symlinks to the portsu ports; e.g.,
 /usr/prtptkg/systems/boxA[rootfsB]/ports/plugh/xyzy -> /usr/prtptkg/CRUX-3.2/ports/plugh/xyzy
 Installed ports that have no port history will be presumed to be associated with the portsu config's normal selection (the highest prtdir having a port for that package).
 Note the /var/ports tree is implicitly connected to the root filesystem in which it resides (and the host thereof), identified elsewhere in the prtptkg tools as 'hostname`\$ROOT where ROOT is set to the enterprise's label for the root filesystem during boot.
 This script takes only optional '-h' and '--help' arguments.

6.25. Prolog: *validate_symports_links* (gawk) [in transition from varports]

from /tmp/validate_symports.links, identify all packages that reside in multiple collections (note pseudo-collection _symlinks ports should never be seen by this script but the logic for dealing with them remains in place). Any multiples found are recorded in /tmp/validate_varports.dups for subsequent remediation by continuing processing of the validate_symports script. Messages of significance are written to stdout, and the status code returned is the number of packages needing remediation

6.26. Prolog: *whatpkg* [in transition to WHATPKG_ variables]

Lookup the port collection and version for installed package \$1 as recorded in the designated prtptkg.txt file.

6.27. Prolog: *whatpkg_2ndline* (gawk) [in transition to WHATPKG_ variables]

from /dev/null, based upon the values of the WHATPKG_PKGINFO and WHATPKG_PRTPKGTX environment variable values, write as appropriate to stdout:
 "pkginfo reports uninstalled"
 --or-- "pkginfo version mismatch: " followed by the value of WHATPKG_PKGINFO
 --or-- nothing
 Regardless, return status code zero is returned.

6.28. Prolog: *whatpkg_pkginfo* (gawk) [in transition to WHATPKG_ variables]

find WHATPKG_PKG environment variable value in input stream from pkginfo -i
 If found, write the installed version to stdout and return status code zero.
 If not found, write "[uninstalled]" to stdout and return status code one.
 Otherwise, write an error message to stdout and return greater than one status.

6.29. Prolog: *whatpkg_prtpkgtxt* (gawk) [in transition to WHATPKG_ variables]

in input `/var/log/prtpkg.txt`, find the most recent successful type B, I, or U record for the package defined in the `WHATPKG_PKG` environment variable, if any. If a suitable record is found, write its following fields separated by tabs:

```
collection port-version buildID batchID tryID
```

and return a zero status code; if one is not found, write an informational message to stdout and return status code one; otherwise, write an error message to stdout and return a status code greater than one.

6.30. Prolog: *whatprt* [in transition to WHATPRT_ variables]

Lookup the port collection for package `$1` that the current `/etc/prt-get.conf` will cause to be selected. Write that as well as the version of the package to stdout and return status code zero, write an informational message to stdout and return status code two if the port is not found, or write an error message to stdout and return a status code greater than two. Status code one is returned when help information is written to stdout.

6.31. Prolog: *whatprt_doit* (gawk) [in transition to WHATPRT_ variables]

from the current `/etc/prt-get.conf` file's `prtdir` statements, determine what port will be selected for processing by `prt-get` and write the collection's path and the `Pkgfile`'s version-release information to stdout. The package to lookup is predefined in the `WHATPRT_PKG` environment variable and `WHATPRT_DBG` is defined as null (no debugging output) or the string to use in the name of the debugging output file; i.e., `/tmp/whatprt.$WHATPRT_DBG.debug`. If the search is successful, status code zero is returned.

- i Usage of the noun system in this document strives to hold to a precise definition that differentiates it from the noun cell that is also precisely defined (later) in this document. A “*system*” is a really or virtually bootable entity that provides more services to the enterprise than merely running prtpkg processes (and perhaps runs no such processes and perhaps not even CRUX) and thus has reliability, availability, security, user data handling/storage, and system administration support requirements beyond those of a mere prtpkg-only bootable entity. In short, prtpkg and its documentation tries to ignore computing outside its mission and focuses on its cells. There is possible overlap when a system runs software maintained by prtpkg, of course, because then it is most likely simultaneously a system and a cell. Also, “*prtpkg platform*” should be understood to be a particular boot cell plus all chroot cells in its root tree that are not mounted in a networked filesystem, and is explicitly meant to *not* include any virtual machine cells hosted via the boot cell.
- ii This endnote explains in one paragraph all the botanical and maritime analogies used to reference filesystem namespace components within this entire document. The noun “*tree*” in this document is an inode in a filesystem that has or can have trees of its own and includes all the referenced tree’s “*branches*” (just *subtrees*) and “*leaves*” (the ends of the lines, never directories or anchors, but possibly reflinks). The noun “*trunk*” connotes the foundational inode of a tree, which must be a directory, possibly empty but having the potential to sprout branches. A symlink to a trunk is not the trunk itself and is referenced using the noun “*anchor*” when referring to the trunk, otherwise it is referred to as a tree. However, the *transitive verb* “*anchor*” implies a direct object that may be a trunk or an anchor. Note that a tree may not necessarily refer to a trunk or an anchor—it may be a file inode in the logical filesystem but its name and any content can identify a tree the logical filesystem should contain somewhere. This works somewhat like a C++ reference so we’ll call this a “*reflink*” when needed. One last definition: “*logical filesystem*” refers to the effective root filesystem including all other filesystems mounted within the root filesystem’s namespace at the time of reference; i.e., all the inodes that are currently accessible via an absolute path.
- iii The noun “*cell*” refers to a prtpkg software building zone. These can be bootable or chrootable; either way, they have unique instances of /etc/ports, /usr/ports, /var, etc.. A system is a prtpkg cell iff it is defined as such in a prtpkg commonwealth. The possibility that the cell is also a system is rarely relevant to the discussion.

- iv The noun “*porter*” refers to a cell that is authorized to perform *portdb* and *portsu* processing within a prtpkg commonwealth (even if the commonwealth is comprised of a single prtpkg platform).
- v The noun “*builder*” refers to a cell that is authorized to perform *pkgmk* processing within a prtpkg commonwealth (even if the commonwealth is comprised of a single prtpkg platform). Builders may be restricted to specific distribution releases and build definitions.
- vi The noun “*deployer*” refers to a cell that is authorized to perform *pkgadd*, *pkgrm*, and *rejmerge* processing on a particular cell (even if the commonwealth is comprised of a single prtpkg platform). Normally deployers are only authorized to add and remove packages on their own BOOTOS ROOTFS tree and may be restricted to packages created for particular distribution releases and build definitions.