ExtUPS— a Tool to Manage Software Dependencies

Nikhil Padmanabhan, Robert Lupton, and Craig Loomis

August 28, 2017

Contents

1 Introduction

Complex software systems are generally built from a hierarchy of components, and in general all of these systems are in a state of flux. In order to bring order out of chaos, we need to be able to:

- Organize the set of components, allowing for their natural hierarchy.
- Select which versions of components should be used, ensuring that a consistent set are chosen
- Configure the environment (`PATH`; environment variables) for each component.
- Provide a way to identify a chosen set of components
- In addition, we would like to be able to install libraries that support these versions, and to do so on a number of different platforms.

We assume that the task of actually managing individual components (e.g. tracking code versions) is being handled with some other system, such as `svn` or `git`.

See Appendix C if you want to know how to install your own copy of `ExtUPS`.

2 The `setup` command

`ExtUPS`[1] achieves all of these goals. A component (refered to as a 'product') may be used after it has been `setup`, and these products may in turn setup other products. When setting up a product, a user may specify a version, or accept the current default. These setup versions may either be in a user's file space (probably a 'live' version checked out from a source code manager), or copies that have been `declare`d to the system; these declarations are aware of the user's computer architecture (referred to as a 'flavor').

The organisation is via 'table' files associated with each component, for example the table file for a product named `photo` reads:

```
setupRequired("astrotools")
envSet(PHOTO_STARTUP, ${PRODUCT_DIR}/etc/photoStartup.tcl)
envAppend(PATH, ${PRODUCT_DIR}/bin)
```

When we attempt to `setup` the product `photo`, this table file leads to the following actions:

The environment variable `PHOTO_DIR` is automatically set to the root of the directory tree for product `photo` (how this is determined is described in the next section)

`setupRequired` states that when you `setup` this product, you should also `setup` a product called `astrotools`. The table file for `astrotools` can (and in fact does) itself `setup` further products. We say that `photo` *depends* on `astrotools`.

`envSet` sets an extra environment variable; here `${PRODUCT_DIR}` refers to the root of this product's tree (i.e. to `PHOTO_DIR` as set above)

`envAppend` appends the directory `${PRODUCT_DIR}/bin` to the current path (there is also a `envPrepend` directive).

So far we have achieved the first of the goals listed in the introduction, describing a hierarchy. The next section describes how to maintain multiple versions of products.

3 Declaring Products and Versions

Whence came `PHOTO_DIR` in the example? From a command of the form [2]

[1] The name comes from `UPS` (`Unix Product Support`), a tool written at Fermi National Accelerator Laboratory, FNAL. `ExtUPS` (pronounced "E-ups", or just "ups", or sometimes even "eeups" as a single word) was basically a reimplementation first in perl and now python of those parts of `UPS` that we have found especially valuable; we have now made extensive extensions.

[2] Actually, a simple `eups declare photo v5_4_25` would suffice

```
eups declare photo v5_4_25 -f Darwin -r /u/products/Darwin/photo/v5_4_25
```

This command updates the `ExtUPS` 'database' (in reality a set of files); this 'database' is then 'queried' by the `setup` command. In the `declare` command,

- `photo` is the name of the product
- `v5_4_25` specifies the version number (by convention this is a tag provided by your source code manager (*e.g.* `svn` or `git`))
- `-f Darwin` specifies the flavor (i.e. my computer's hardware and operating system). You should almost always omit an explicit flavor declaration; it's not needed and you can get it wrong.
- `-r /u/products/Darwin/photo/v5_4_25` is the directory where the specified version can be found. The name is purely conventional, but the convention is pretty transparent.

  This directory is typically created by an `install` target in a Makefile, but it could simply be your current working copy: [3]

  ```
  eups declare photo cvs -r ~/photo
  ```

If I now say `setup photo v5_4_25`, `ExtUPS` will look for a table file in the directory `/u/products/Darwin/photo/v5_4_25/ups`, and obey whatever instructions it finds there. For example, I will find that `PHOTO_DIR` is set to `/u/products/Darwin/photo/v5_4_25`. If, in the `declare` command I'd also specified `-c` (for 'current'), then version `v5_4_25` of `photo` would have become the default, and I could have simply said `setup photo`.

You will remember that `photo`'s table file simply requested that `astrotools` be setup, but it could have specified a version:

```
setupRequired("astrotools v4_10_1")
```

The table file being used is the one in the *installed* directory, and the `make install` command was free (and indeed encouraged) to rewrite the table file to specify explicit version numbers using the `eups expandtable` command.

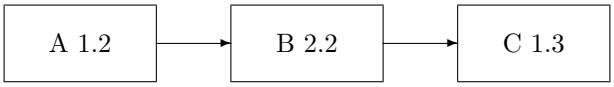
We have now achieved all of our five goals (although we have not yet described how to make use of our success): I have defined a version `v5_4_25` of a product `photo`, which is to be found in a given directory for a given architecture, and specified how its environment should be configured. Furthermore, this version depends on a specific version (`v4_10_1`) of `astrotools`, and so on. By convention, each of these version numbers corresponds to a tag in some source code repository, so I am able to recover the complete set of releases used to build and run this version of `photo` by examining the `ExtUPS` database.

4 The `setup` command continued

4.1 How `ExtUPS` decides which version of a package to setup

We rather blithely described how `ExtUPS` chooses a version with the phrase, "When setting up a product, a user may specify a version, or accept the current default". Unfortunately, it isn't always quite so simple.

Consider a simple "dependency tree" of products such as:



i.e. version 1.2 of product A sets up version 2.2 of B which sets up version 1.3 of C. This chain is defined by the product's table files; i.e.. A 1.2's table looks something like:

[3] It's probably better to use a tag (Sec. 4.3) here: `eups declare -t rhl -r .`

```
setupRequired(B 2.2)
```

and B 2.2's table:

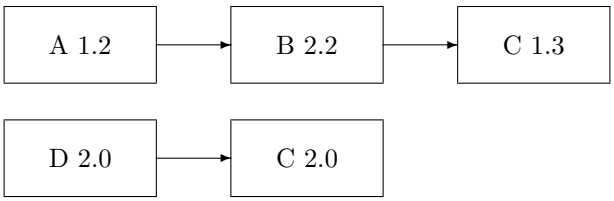
```
setupRequired(C 1.3)
```

When we say `setup A 1.2` setup's task is simple; it looks up A's table file and finds `setupRequired(B 2.2)`; so it sets up B 2.2 finding `setupRequired(C 1.3)`, so it sets up C 1.3 and that's it (as C has no dependencies).

`ExtUPS` can also handle `setup A` (i.e. no explicit version). In this case it looks for the version of A that has been tagged 'current'; if A 1.2 is current, then `setup A` has the same result as `setup A 1.2`. [4]

Unfortunately even specifying explicit versions of dependent products doesn't completely define which products should be setup.

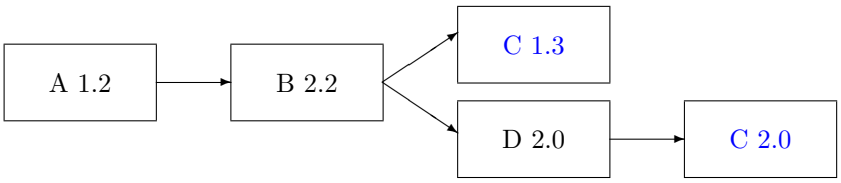
Consider



If we change B 2.2's table file to read

```
setupRequired(C 1.3)
setupRequired(D 2.0)
```

i.e. if we add a dependency on D version 2.0 to B version 2.2, we have:



Note that there are two separate requests for C; this is referred to as a "diamond dependency" by analogy to multiple inheritance in OO languages. `ExtUPS` is unable to handle this without help. One solution is that `eups expandtable` writes the complete set of setup products into the table file (see Sec. B.3.6 for details) and when you specify an exact version or use the `--exact` flag this set of versions is used. For example, if C 1.3 was setup when you installed and declared A 1.2, then the command `setup A 1.2` would result in A 1.2, B 2.2, D 2.0, and C 1.3 being setup.

Another possibility is to always accept the latest version of a product; this is supported but has proved to be fragile as it turns out to be hard to define an absolute order for versions.

Another way to get exactly your desired mix-n-match of versions is to use tags; tagging your favourite version of C preempts any other choice (see Sec. 4.3).

There are a couple of other `setup` options that can be used to control dependencies. One is `--just` which is equivalent to `--max-depth 0`; that is, don't setup any dependencies (so `setup --just A 1.2` wouldn't setup B, whereas `setup --max-depth 1 A 1.2` would setup B but not C or D.

Another is `--keep` which means that any products that you've already setup are preserved; e.g. after `setup C 1.0; setup --keep A 1.2` would result in A 1.2, B 2.2, D 2.0, and C 1.0 being setup.

[4] You can actually change this behaviour to make any tag the default by changing the VRO; see Sec. 4.2

4.2 The VRO

When looking for a version of a product, `setup` searches a list known as the VRO (the `V`ersion `R`esolution `O`rder) looking for a valid recipe to find a version; you can see the current vro with `eups vro`. Some entries on the VRO are directives (e.g. `keep`), while others tell `ExtUPS` how to look for a product (e.g. `version`).

For example, a VRO of `version current` means that when looking for a product, `ExtUPS` should first try to an explicit version specification, and only then a version with tag `current` (it is a VRO like this that makes `current` be the default version). That is, if foo 6.6.6 is current and we see a table file such as

```
setupRequired(foo 1.2.3)
```

a VRO of `version current` would use `version` to setup foo 1.2.3. If the VRO had been `current version` then foo 6.6.6 would have been setup instead.

As the VRO is processed, entries are interpreted as follows:

current Use a version tagged `current`

commandLineVersion Use the version explicitly specified on the command line

keep If a version of the product is already setup, accept it as the new version

latest Use the most recent version [not well tested; caveat usor]

path Use the version specified by an explicit path on the command line (i.e. `-r ...`)

tagname Use the version with a specific tag (see Sec. 4.3)

type:*typename* Set the build type, exactly as if you'd said `--type` *`typename`*. This directive is seen when this element of the VRO is processed, so you probably want to put it first. Its major use is to set `type:exact`; this is the the mechanism used to make commands like `setup foo 1.2` behave as if you'd typed `setup --exact foo 1.2`.

version Use a version of the product specified explicitly (e.g. `1.2.3`)

versionExpr Use a version that satisfies an expression (e.g. `>= 1.2.3`) (explicit versions are allowed too, and imply equality)

warn Print a warning message if the version resolution reaches this point in the VRO without finding a version. You can add a verbosity level (e.g. `warn:2`) to specify the minimum verbosity to print the message.

The VRO can be set by the `--vro` option, or (more commonly) by setting `hooks.config.Eups.VRO` in a startup file (Sec. B.2). `hooks.config.Eups.VRO` is a dictionary with keys that specify the VRO to use with a commandLineVersion, with a tag (e.g. `rhl`) or a default value (key `"default"`); the default can itself be a dictionary with keys corresponding to elements of the `$EUPS_PATH` (cf. the `-z` option). The `eups vro` command can be used to show what VRO would be used (e.g. `eups vro -tag rhl afw 1.2`)

4.3 Tags

It's possible to associate a name with a set of versions; tags are specified as `-t XXX` or `--tag XXX`. After tagging a variety of products as `rhl`, I can say `setup -t rhl photo` to get a version of the product `photo` setup using dependencies that I've honoured with the tag `rhl`; other products are found on the VRO as usual.

Tags may be set with a command such as `eups declare -t rhl afw 1.2.3` or `eups declare -t rhl -r .`[5] There are two sorts of tags; user tags are only available to you, while global tags are visible to all

[5] yes; no version name is required as the tagname suffices to identify the product's version.

users of your ExtUPS database.[6] Additionally, tags may be labelled as 'reserved' which means that you're not supposed to meddle with them; they are designed for centrally-assigned tags such as 'pre-release' or 'golden'.

Tags must be declared before use (to avoid typos such as `-t rkl` when you obviously meant `-t rhl`). This is done in one of the startup files (see Sec. B.2), typically `$HOME/.eups/startup.py` with code such as

```
hooks.config.Eups.globalTags += ["test"]
hooks.config.Eups.userTags += ["t1570"]
```

This declares tags `test` and `t1570`. In addition, your username (in my case `rhl`) is automatically defined as a valid user tag.

You can list all known tags with `eups tags`; user tags are prefixed `user:`. For example,

```
$ eups tags
current latest stable user:rhl
```

When using a tag, the tag is automatically inserted near the front of the VRO;

```
$ eups vro
path versionExpr warn current
$ eups vro -t rhl
path rhl versionExpr warn current
```

In other words, your tagged products take priority over any other version specification (except an explicit path). If you'd rather give priority to version expressions over your tags, specify them with `-T` or `--postTag`:

```
$ eups vro -T rhl
path versionExpr rhl warn current
```

If you need even more complete control, use the `--vro` command.

Post-tags are useful when you want exact products, but you don't want to have to search your email for the name of currently-blessed version, e.g. `afw 6.1.3.1`. You might think of tagging it with `rhl`: `eups declare -t rhl afw 6.1.3.1; setup -t rhl afw` and that would work until you tagged some dependent product `rhl` — at that point you'd get both of your tagged products. The solution is to make the tag only apply if no version is known, in other words put the tag *after* `version` in the `vro`. And this is exactly what `-T rhl` does.

As an alternative declaring a version with a known tagname, the 'tagname' in `-t` *tagname* may be a file consisting of pairs `product version ...` such as that produced with the `eups list --setup` command [7]

For example, to share a setup with a colleague I could say:

```
$ eups list --setup > mysetups.txt
$ emacs mysetups.txt       # remove products I don't care about
$ setup -t mysetups.txt afw
```

and then tell her to repeat that last line. If she wants to use some of her bug-fixed versions in conjunction with my versions, she'd say `setup -t lynda -t mysetups.txt afw` which is why this approach is often better than just sourcing a set of `setup -j XXX` lines.

4.3.1 Specifying default Tags

If you don't specify any tags to the `setup` (or `eups vro`) command (with either `--tag` (`-t`) or `--postTag` (`-T`)) then `hooks.config.Eups.defaultTags` is consulted. It's a dictionary with keys `pre` and `post`; e.g. if you have

```
hooks.config.Eups.defaultTags["pre"] += ["rhl", "HSC"]
hooks.config.Eups.defaultTags["post"] += ["v6_1"]
```

in `startup.py`, then `setup foo` is exactly equivalent to `setup -t rhl,HSC -T v6_1`. You can disable this default tag processing with `--tag None` (or `--tag ""`).

[6] You are permitted to use other people's user tags if you can read their `~/.eups` directory; e.g. to use my beta tag put `hooks.config.Eups.userTags += [("beta", "rhl")]` into *your* `~/.eups/startup.py` file

[7] You may also say `file:XXX` to explicitly specify that `XXX` should be interpreted as a file, not a tagname

5 How do I Actually Use this System to Manage Code?

All that the `setup` command did was to set or append to a number of environment variables, such as `PHOTO_DIR`, `ASTROTOOLS_DIR`, and (often) `LD_LIBRARY_PATH`. These can be used in Makefiles, e.g.

```
CFLAGS = -g $(INCS)
INCS = -I$(ASTROTOOLS_DIR)/include \
       -I$(PHOTO_DIR)/include \
       ...
LIBS =   -L$(ASTROTOOLS_DIR)/lib -lastrom \
         -L$(PHOTO_DIR)/lib -lphoto -lmeasureObj \
         ...
```

App. E discusses a strategy for managing include files and libraries using configure and make; but the gnu build tools are complicated and (some would say) confusing and clunky, so you may not want to learn them. One option is to add some more magic to the Makefile (see the next section); another is to give up entirely on make and use some more sophisticated build tool. The one that we have experience with is `scons`, and a complete integration of `ExtUPS` with scons is described in section `scons`.

5.1 `ExtUPS` and make

This is all very well, and perfectly practical, but it doesn't integrate very closely with `ExtUPS`; in particular it doesn't help you much with installing in Proper Places. If you are using GNU's version of make (and you probably are), you can handle this by a set of rules looking something like:

```
PRODUCT = test
EUPS_ROOT = $(shell eups path -1)
VERSION = $(shell ./eups_product_version)
FLAVOR = $(shell eups flavor)

ifeq (, $(prefix))
   prefix = $(EUPS_ROOT)/$(FLAVOR)/$(PRODUCT)/$(VERSION)
endif

install :
        install -m 775 bin/foo $(prefix)/bin
        eups expandtable ups/$(PRODUCT).table $(prefix)/ups

declare :
        eups declare -r $(prefix) $(PRODUCT) $(VERSION)

current : declare
        eups declare $(PRODUCT) $(VERSION) --current --force
```

`EUPS_ROOT` is taken to be the last element in `EUPS_PATH` (that's the `-1` in `eups path -1`), and the `ifeq` bit allows me to say `make install declare version=/usr/local`.

The script `eups_product_version` (and this Makefile) are to be found in the `ExtUPS etc` directory, although they are not installed along with `ExtUPS`.

The script `eups_product_version` is rather like:

```
#!/usr/bin/env python
import eups

print eups.version('$Name: $')
```

where the `$Name: $` is a magic cvs keyword that expands to your current tagname, so you need to put a copy of the script in your own package, so as to get *your* tagname.

If you use svn, things are a bit harder as svn provides no real equivalent to a tagname. Simpilar functionality can be achieved using `$HeadURL$` and requiring that all tags be in a directory `tags` or `TAGS` within the svn repository.[8] The real `eups_product_version` attempts to correctly handle `cvs`, `svn`, `hg` and `git` users.

5.2 `ExtUPS` and scons

Write me.

A Using `ExtUPS` with python

`ExtUPS` is written in python, so you can use it in scripts by simply importing it.

To use `ExtUPS` from python, you need to `import eups`; The functions in `eups.app` define the supported API:

clearCache remove the product cache for given stacks/databases and flavors

declare Declare a product. That is, make this product known to EUPS.

expandBuildFile expand the template variables in a .build script to produce an explicitly executable shell scripts.

expandTableFile expand the version specifications in a table file. When a version in the original table file is expressed as an expression, the expression is enclosed in brackets and the actual product version used to build the table file's product is added.

getDependencies Return a list of productName's dependent products

getDependentProducts Return a list of Product topProduct's dependent products

getSetupVersion return the version name for the currently setup version of a given product.

listCache List the contents of the cache

printProducts print out a listing of products. Returned is the number of products listed.

printUses print a listing of products that make use of a given product.

productDir return the installation directory (PRODUCT_DIR) for the specified product

setup Return a set of shell commands which, when sourced, will setup a product.

undeclare Undeclare a product. That is, remove knowledge of this product from EUPS. This method can also be used to just remove a tag from a product without fully undeclaring it.

unsetup Return a set of shell commands which, when sourced, will unsetup a product.

[8] and saying `svn propset svn:keywords HeadURL`

B ExtUPS Reference Manual

B.1 ExtUPS concepts and jargon

current The special *tag* used to indicate the default version of a *product*.

dependency A *product* B is a dependency of A if B needs to be setup as part of setting up A.

flavor The computer architecture for which your *product*s are valid. This is often more-or-less a summary of your `uname`; examples are `DarwinX86`, `Linux`, `Linux64`.

product A logical unit that you would like to version using `ExtUPS`

root The directory associated with a *product* (if any). For the common case of a *product* containing source code the root would usually contain include and library subdirectories, although this is not required.

tag A name attached to a set of one or more *products*, which can be used to override the version that would otherwise be selected. Note that this use of the word *tag* is unrelated to its use in source code managers such as `svn` or `hg`.

table A *product*'s *table file* lists the actions that `ExtUPS` needs to carry out when setting up the *product*.

version A string specifying which version of a *product* is being used. Conventionally these correspond to names generated from your source code manager (e.g. `svn`), but this is not required.

VRO The `V`ersion `R`esolution `O`rder; the order in which possible versions of a *product* are considered.

B.2 How to customize ExtUPS

You can customize `ExtUPS`'s behaviour in two ways:

- By setting an environment variable `$EUPS_STARTUP` which points to a python script
- By putting a `startup.py` in a well known place, usually `$HOME/.eups` but in fact `ExtUPS` searches a number of place in order.

You can list your startup files with `eups startup`; with `-v` files that would be consulted if they are existed are also listed. [9]

For example if `$EUPS_STARTUP=$HOME/Python/foo.py` and `$EUPS_PATH=/u/lsst/products:/u/lsst/coreProducts`, then `eups -v startup` reports

```
/u/lsst/products/eups/site/startup.py
/u/lsst/products/site/startup.py
[/u/lsst/coreProducts/site/startup.py]
/Users/rhl/.eups/startup.py
/Users/rhl/Python/foo.py
```

These files are processed in the order listed (i.e. `$EUPS_STARTUP` last)

[9] Once upon a time there were two sorts of startup file: `startup.py` and `config.properties`. The latter are currently disabled in `hooks.py`

B.2.1 Contents of startup files

Hook commands There are a number of variables that can be set using syntax such as
`hooks.config.Eups.reservedTags += ["stable"]`

userTags Permitted user tags. See Sec. 4.3.

preferredTags XXX

defaultTags A dict indexed by `pre` and `post`. See Sec. 4.3.1.

globalTags Defined global tags. See Sec. 4.3.

reservedTags

verbose

asAdmin

setupTypes Permitted arguments to `setup --type=XXX`, *e.g.* `hooks.config.Eups.setupTypes.append("sdss")`

setupCmdName

VRO

fallbackFlavors

For example, I have

```
hooks.config.Eups.globalTags += ["test"]
hooks.config.Eups.userTags += ["t1570"]
hooks.config.Eups.reservedTags += ["stable"]
hooks.config.Eups.VRO["default"]["dss"] = "type:exact path version current"
```

B.2.2 Callback functions

You can also register callback functions which are called twice; one just before the main initialisation (the one that creates the `Eups` object) and once just before `ExtUPS` settles down to work.

The arguments to the callbacks are:

`Eups` An instance of the class `Eups` (or `None` the first time the callback is called, before the `Eups` object has been created)

`cmd` The command being executed (e.g. `setup` or `list`)

`opts` The options being passed to the command

`args` The arguments being passed to the command

For example, you could put something like this in a `startup.py` file:

```
import eups

def cmdHook(Eups, cmd, opts, args):
    import eups

    if not Eups:
        if cmd in ("setup", "unsetup"):
            eups.enableLocking(False)
        return

    if Eups and cmd == "setup":
        if not opts.tag:              # see also config.Eups.defaultTags
            opts.tag = ["rhl", "Winter2012c", "beta"]

            if opts.verbose >= 0:
                import utils
                print >> utils.stdinfo, "Adding default tags: %s" % (", ".join(opts.tag))
eups.commandCallbacks.add(cmdHook)
```

(Since this was written a configuration option, `hooks.config.Eups.defaultTags`, was added to support default tags without writing a callback.)

B.2.3 Colour

It is possible to get `ExtUPS` to colour its error, warning, and informational messages if your terminal co-operates. By default color is turned off; to enable it put a line

```
hooks.config.Eups.colorize = True
```

in your startup file. If you don't like our choice of colours, you can instead say *e.g.*:

```
hooks.config.Eups.colorize = dict(ERROR = "red", WARN = "blue", INFO = "cyan")
```

Available colours are `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, and `white`. Additionally, you may add `;bold` (*e.g.* `red;bold`).

Colour is only used if your `stderr` is going to a terminal.

B.3 ExtUPS commands

B.3.1 eups

```
Usage:
        eups [--help|--version] command [options]
Supported commands are:
        admin           Administer the eups system
        declare         Declare a product
        distrib         Install a product from a remote distribution,
                        or create such a distribution
        expandbuild     Expand variables in a build file
        expandtable     Insert explicit version tags into a table file
        flags           Show the value of \$EUPS_FLAGS
        flavor          Return the current flavor
        help            Provide help on eups commands
        list            List some or all products
        path [n]        Print the current eups path, or an element thereof
        pkgroot [n]     Print the current eups pkgroot, or an element thereof
        pkg-config      Return the options associated with product
        remove          Remove an eups product from the system
        tags            List information about supported and known tags
        undeclare       Undeclare a product
        uses            List everything which depends on the specified product
                        and version
        vro             Show the Version Resolution Order that would be used

Use
        eups --help cmd
for help with command "cmd"
```

A consistent front-end to all the other ExtUPS commands.

There's no `eups setup` or `eups unsetup` as these actions modify your environment and are in fact aliases for commands that source files of `export ...` (or `setenv ...`) lines into your current shell.

All `eups` commands support a common set of options:

```
     --debug      arg    Permitted Values: raise
  -f, --flavor    arg    Use this flavor. Default: $EUPS_FLAVOR or ‘eups flavor‘
  -F, --force            Force requested behaviour
  -h, --help             Print this help message
  -n, --noaction         Don’t actually do anything
  -z, --select-db arg    Select the product paths which contain this directory.
                         Default: all
  -Z, --database  arg    Use this products path. Default: $EUPS_PATH
                         Alias: --with-eups
  -v, --verbose          Be chattier (repeat for even more chat)
  -V, --version          Print eups version number and exit
  --vro=LIST             Set the Version Resolution Order
```

The flavor is usually determined by the command `$EUPS_DIR/bin/eups flavor`, but may be set using the environment variable `$EUPS_FLAVOR`, or `--flavor` (in order from lowest to highest precedence).

The desired database can also be specified explicitly with the `-Z` option (which overrides `EUPS_PATH`), or by using `-z` to specify a path component; e.g. `-Z /home/proj1/eups:/home/proj2/eups -z proj2` would select `/home/proj2/eups`.

B.3.2 `eups admin`

```
Usage:
    eups admin [options] [buildCache|clearCache|listCache|clearLocks|clearServerCache|info|show]

Options:
   -r, --root       arg    Location of manifests/buildfiles/tarballs (may be a URL or scp specificatio
                           Default: $EUPS_PKGROOT
```

There are a number of subcommands:

- `buildCache|clearCache|listCache` Manipulate `ExtUPS`'s caches
- `clearServerCache` Clear the `eups distrib` cache
- `clearLocks` Clear `ExtUPS` locks
- `info product [--tag XXX] [version]` Provide information about a product, specifically the location of the file that `ExtUPS` (currently) uses to define a product/version/tag.
- `show` Show the value of something of interest to `ExtUPS`.

B.3.3 `eups declare`

```
Usage:
   eups [commonOptions] declare [options] [product [version]]
Options:
   -c, --current           Declare product current
   -L, --import-file arg   Import a file directly into the database
   -M                arg   Import the given table file directly into the database
                           (may be "-" for stdin)
   -m, --table       arg   Use table file (may be "none") Default: product.table
   -r, --root        arg   Location of product being declared
   -t, --tag         arg   Install versions with this tag (e.g. current or stable)
```

Declares `version` of `product` to `ExtUPS`.

The `eups declare` command has to decide which database in the environment variable `EUPS_PATH` to use. If the product's already declared, use the database it's in (this is usually only relevant when you're declaring a product current). Otherwise, if the specified root lies within a database in `EUPS_PATH`, use that database; as a last resort use the database that's listed first in `EUPS_PATH`.

You can use `-L fileName` to tell `ExtUPS` to save you a copy of `fileName` in the 'extra product directory' (usually because you are declaring a product that you don't have write permission for). When you setup a product that was declared this way, an extra environment variable `PRODUCT_DIR_EXTRA` is set to point to this directory; it's also available as `${PRODUCT_DIR_EXTRA}` in table files.

The argument to `-L` is in fact a bit more flexible:

`fileName`	Save `fileName` as `$PRODUCT_DIR_EXTRA/fileName`
`fileName:outName`	Save `fileName` as `$PRODUCT_DIR_EXTRA/outName`
`fileName:dirName/`	Save `fileName` as `$PRODUCT_DIR_EXTRA/dirName/fileName` (note the trailing `/`)

The `-m` option specifies the table file to be used (either in the database, or in the ups directory of the product). Your permitted options are actually only `product.table` or `none`; but see `-M`.

The `-M` option specifies a file which will be copied into the database as a table file. This lets you avoid putting EUPS data into products. You can pipe the table file to `eups declare` by specifying `-` as the filename. The option `-M bar` is equivalent to `-L bar:ups/product.table`, *i.e.* the table file may be found in `$FOO_DIR_EXTRA/ups` (you can also find it with `eups list -m foo`).

If you omit the `-r dir`, `ExtUPS` will attempt to guess it for you based on your `$EUPS_PATH`, flavor, product and version; alternatively, if you omit both `product` and `version`, but provide `-r dir`, the product and version will be guessed from the last two parts of the directory name (e.g. `-r /home/products/Linux/xpa/v2_1` will be taken to be declaring version `v2_1` of product `xpa`). If you only omit the product, the product will be guessed from the name of the table file present in the root directory. For example, the following are all equivalent:

```
eups declare -r /u/lsst/products/DarwinX86/foo/1.2.3 foo 1.2.3
eups declare -r /u/lsst/products/DarwinX86/foo/1.2.3 1.2.3
eups declare -r /u/lsst/products/DarwinX86/foo/1.2.3
eups declare foo 1.2.3
```

(assuming that the file `/u/lsst/products/DarwinX86/foo/1.2.3/ups/foo.table` exists). You may need to specify the version, e.g.

```
eups declare -r /u/rhl/foo
```

tries to declare version `foo` of product `rhl`; this will fail as there is presumably no file `/u/rhl/foo/ups/rhl.table`. You needed to say

```
eups declare -r /u/rhl/foo myVersion
```

You may specify `-r none`, in which case you don't have any files; this is only permitted if you also specify `-M` to provide a table file (or `-m none`). This can be handy if you want an alias; e.g.

```
echo "setupRequired(visionWorkbench)" | eups declare -r none -M - vw current
```

makes `vw` an alias for product `visionWorkbench`. You can make an alias for a specific version with e.g.

```
setup visionWorkbench 1.0.0
echo "setupRequired(visionWorkbench)" | eups expandtable | \
eups declare -r none -M - vw 1.0.0
```

If you want to know what `vw` does, type e.g.

```
cat $(eups list vw -m)
```

If you specify `-c` then it leaves the currently declared version alone, and only updates the current.chain. If you are declaring the first version of a new product it's automatically made current.

B.3.4 eups distrib

```
Usage: eups distrib [clean|create|declare|install|list|path] [-h|--help] [options] ...

Interact with package distribution servers either as a user installing
packages or as a provider maintaining a server.

An end-user uses the following sub-commands to install packages:
   list      list the available packages from distribution servers
   path      list the distribution servers
   install   download and install a package
   clean     clean up any leftover build files from an install (that failed)
To use these, the user needs write-access to a product stack and database.

A server provider uses:
   create    create a distribution package from an installed product
   declare   declare global tags
To create packages, one must have a write permission to a local server.

Type "eups distrib [subcmd] -h" to get more info on a sub-command.

Common
Options:
  --debug=DEBUG         turn on specified debugging behaviors (allowed: debug,
                        profile, raise)
  -h, --help            show command-line help and exit
  --noCallbacks         Disable all user-defined callbacks
  -n, --noaction        Don't actually do anything (for debugging purposes)
  --nolocks             Disable locking of eups's internal files
  -q, --quiet           Suppress messages to user (overrides -v)
  -T SETUPTYPE, --type=SETUPTYPE
                        the setup type to use (e.g. exact)
  -v, --verbose         Print extra messages about progress (repeat for ever
                        more chat)
  -V, --version         Print eups version number
  --vro=LIST            Set the Version Resolution Order
  -Z PATH, --database=PATH
                        The colon-separated list of product stacks (databases)
                        to use. Default: $EUPS_PATH
  -z DIR, --select-db=DIR
                        Select the product paths which contain this directory.
                        Default: all in path
  --with-eups=PATH      synonym for -Z/--database
  -f FLAVOR, --flavor=FLAVOR
                        Assume this target platform flavor (e.g. 'Linux')
  -F, --force           Force requested behaviour
```

Handle the distribution of products and their dependencies as 'packages'. A package is a set of tarballs (i.e. gzipped tar archives), a set of build files[10] or pacman[11] caches and a file (ending in `.manifest`) listing a product's dependencies, including the names of the proper tarballs/caches.

[10] see Appendix F.2.
[11] See Appendix F.

Subcommand options are:

- clean

```
eups distrib clean options product version

Options:
   -n, --noaction          Don't actually do anything
   -R, --remove            Clean and Remove all remnants of a declared product
   -r, --root       arg    Location of manifests/buildfiles/tarballs (may be a URL or scp specifica
                           Default: $EUPS_PKGROOT
   -S, --server-class arg  register this DistribServer class (repeat as needed)
   -t, --tag        arg    Install versions with this tag (e.g. current or stable)
```

- create

```
eups distrib create options product [version]

Options:
  -a PACKAGEID, --as=PACKAGEID
                        Create a distribution with this name
  -d DISTRIBTYPENAME, --distribType=DISTRIBTYPENAME
                        Create a distribution with this type name (e.g.
                        'tarball', 'builder')
  -I, --incomplete      Allow a manifest including packages we don't know how
                        to install
  -m MANIFEST, --manifest=MANIFEST
                        Use this manifest file for the requested product
  -j, --nodepend        Just create package for named product, not its
                        dependencies
  -r BASEURL, --repository=BASEURL
                        the base URL for other repositories to consult (repeat
                        as needed).  Default: $EUPS_PKGROOT
  -s DIR, --server-dir=DIR
                        the directory tree to save created packages under
  --flavor=FLAVOR       Assume this target platform flavor (e.g. 'Linux')
  -F, --force           Force requested behaviour
  -D DISTRIBCLASSES, --distrib-class=DISTRIBCLASSES
                        register this Distrib class (repeat as needed)
  -R REBUILDPRODUCTVERSION, --rebuild=REBUILDPRODUCTVERSION
                        Create a new distribution given that product:version's
                        ABI has changed
  --rebuildSuffix=REBUILDSUFFIX
                        Specify suffix to apply to new versions generated by
                        --rebuild
  --server-class=SERVERCLASSES
                        register this DistribServer class (repeat as needed)
  -S SERVEROPTS, --server-option=SERVEROPTS
                        pass a customized option to all repositories (form
                        NAME=VALUE, repeat as needed)
  -e, --exact           Follow the as-installed versions, not the dependencies
                        in the table file
  -f USEFLAVOR, --use-flavor=USEFLAVOR
                        Create an installation specialised to the specified
                        flavor
  -T POSTTAG, --postTag=POSTTAG
                        Put TAG after version(Expr)? in VRO (may be repeated;
  -t TAG, --tag=TAG     Set the VRO based on this tag name
  --debug=DEBUG         turn on specified debugging behaviors (allowed: debug,
                        raise)
  -h, --help            show command-line help and exit
  -n, --noaction        Don't actually do anything (for debugging purposes)
  --nolocks             Disable locking of eups's internal files
  -q, --quiet           Suppress messages to user (overrides -v)
  --type=SETUPTYPE      The setup type to use (e.g. exact)
  -v, --verbose         Print extra messages about progress (repeat for ever
                        more chat)
  -V, --version         Print eups version number
  --vro=LIST            Set the Version Resolution Order
  -Z PATH, --database=PATH
                        The colon-separated list of product stacks (databases)
                        to use. Default: $EUPS_PATH
  -z DIR, --select-db=DIR
                        Select the product paths which contain this directory.
                        Default: all in path
  --with-eups=PATH      synonym for -Z/--database
```

- `declare`

  ```
  Usage: eups distrib declare [-h|--help] [options] [product [version]]

  Declare a tag for an available package from the package distribution
  repositories.  If no product or version is provided, all defined tags are
  defined.

    ...
  ```

  Declare a tag which `eups distrib install` will make use of, e.g.

  ```
  eups distrib declare --server-dir=$HOME/packages -t current afw
  ```

  You can disable the installation of the tag with `eups distrib install --no-server-tags ...`.

- `install`

```
usage: eups distrib install [-h|--help] [options] product [version]

Install a product from a distribution package retrieved from a repository. If
a version is not specified, the most version with the most preferred  tag will
be installed.

options:
  -d TAG, --declareAs=TAG
                        tag all newly installed products with this user TAG
                        (repeat as needed)
  -g GROUP, --groupAccess=GROUP
                        Give specified group r/w access to all newly installed
                        packages
  -I DIR, --install-into=DIR
                        install into this product stack (Default: the first
                        writable stack in $EUPS_PATH)
  -m MANIFEST, --manifest=MANIFEST
                        Use this manifest file for the requested product
  -U, --no-server-tags  Prevent automatic assignment of server/global tags
  --noclean             Don't clean up after successfully building the product
  -j, --nodepend        Just install product, but not its dependencies
  -N, --noeups          Don't attempt to lookup product in eups (always
                        install)
  -r BASEURL, --repository=BASEURL
                        the base URL for a repository to access (repeat as
                        needed).  Default: $EUPS_PKGROOT
  -t TAG, --tag=TAG     preferentially install products with this TAG
  --tmp-dir=DIR         Build products in this directory
  --nobuild             Don't attempt to build the product; just declare it
  -f FLAVOR, --flavor=FLAVOR
                        Assume this target platform flavor (e.g. 'Linux')
  -F, --force           Force requested behaviour
  -D DISTRIBCLASSES, --distrib-class=DISTRIBCLASSES
                        register this Distrib class (repeat as needed)
  --server-option=SERVEROPTS
                        pass a customized option to all repositories (form
                        NAME=VALUE, repeat as needed)
  -S SERVERCLASSES, --server-class=SERVERCLASSES
                        register this DistribServer class (repeat as needed)
  --debug=DEBUG         turn on specified debugging behaviors (allowed: raise)
  -h, --help            show command-line help and exit
  -n, --noaction        Don't actually do anything (for debugging purposes)
  -q, --quiet           Suppress messages to user (overrides -v)
  --type=SETUPTYPE      The setup type to use (e.g. exact)
  -v, --verbose         Print extra messages about progress (repeat for ever
                        more chat)
  -V, --version         Print eups version number
  --vro=LIST            Set the Version Resolution Order
  -Z PATH, --database=PATH
                        The colon-separated list of product stacks (databases)
                        to use. Default: $EUPS_PATH
  -z DIR, --select-db=DIR
                        Select the product paths which contain this directory.
                        Default: all in path
  --with-eups=PATH      synonym for -Z/--database
  --recurse             don't assume manifests completely specify dependencies
  --root=ROOT           equivalent to --repository (deprecated)
  -C, --current-all     Disabled; make all products we install current
  -c, --current         Make top level product current (equivalent to --tag
                        current)
```

Usually `--force` will reinstall all products, but you can override this with an entry in `manifest.remap`:

```
product[:version]  noReinstall
```

and is useful if you have local declarations of e.g. python.

- `path`

```
Usage: eups distrib path [-h|--help] [n]

Print the base URLs for the repositories given via EUPS_PKGROOT.  An
optional integer argument, n, will cause just the n-th URL to be listed
(where 0 is the first element).

Options:

  ...
```

You may use `eups distrib list` to list available products; with `--tag current` only current products are listed. E.g.

```
eups distrib list --root http://www.astro/~rhl/LSST/packages --tag current mwi 2.4
```

(or you could use any other tag of your choice).

Once you've created a package with `eups distrib create` you can log on to another machine and use `eups distrib install` to recreate a working `ExtUPS` installation. For example:

```
eups distrib --verbose --select-db act create moby v1_2
eups distrib --database /u/rhl/ppp --verbose install \
             --root http://www.astro.princeton.edu/~act/packages moby v1_2
```

to install `moby` into a `/u/rhl/ppp` `ExtUPS` database. If you omit `--root` it'll be taken from the environment variable `EUPS_PKGROOT`, if set. As an alternative to an http URL, you may specify a directory as if you were using `scp`, prefixed by `scp:`; e.g.

```
--root=scp:rhl@apache2.astro.princeton.edu:WWW/public/distrib
```

When creating an distribution, tarballs that already present are not recreated (unless you specify `--force`), so repeating a `eups distrib create` command to create a manifest for another product won't lead to duplication of tarballs. If you want to use buildfiles see App. F.2; if you'd like to try `pacman`, see App. F.

When unpacking a distribution with `eups distrib install`, products that are already declared to `ExtUPS` won't be unpacked and re-declared (unless you specify `--force`).

If you don't specify a version when installing a distribution with `--install` the current version will be used. This isn't quite the same as the version declared current on some machine where the distribution originated, as a bleeding-edge version acceptable in Princeton may not be ready for deployment on a mountain top in e.g. Chile. In consequence, the list of 'current' versions is *not* automatically created or updated by `eups distrib create`, but is only set when you later say `eups distrib create --current`. If you don't specify a product, the list will in fact be identical to your local preferences. Alternatively, you can update a single product's version number (e.g. `eups distrib create --current moby`) or specify a version (`eups distrib create --current moby v1_2`).

N.b. As presently written, `eups distrib` assumes that (unless you're using `--noeups`) you've installed all the products into `ExtUPS`, as opposed to simple declaring them, but relying on files installed in e.g.

/usr/local; more precisely, it assumes that all product directories start with a directory listed in EUPS_PATH. This is probably a good idea, as otherwise eups distrib would tar up all of /usr/local for you.

N.b. A decent description of the -d builder -S buildFilePath=D:D: version of this command needs to be written! This would include the -S noeups=True option, and its use of product.table files in the BuildFilePath

For example, to create a distribution of product moby (already declared to ExtUPS on my machine), install it in a central place (on apache2)), and then install in on some other machine I first created a build file moby.build in moby's ups directory:

```
export CVSROOT="@CVSROOT@"

cvs co -P -d @PRODUCT@-@VERSION@ -r @VERSION@ @PRODUCT@ &&
cd @PRODUCT@-@VERSION@ &&
setup -r . &&
./configure &&
make &&
make install
```

and then typed:

```
eups distrib create -v -s /u/act/products/packages -d builder -S buildFilePath=~/ACT//BuildFiles:$EUPS
rsync -rv /u/act/products/packages apache2:/u/act/WWW/public

eups distrib -v install --current moby --root http://www.astro.princeton.edu/~act/packages
```

The /ACT/BuildFiles: tells ExtUPS to first look for build files in /ACT/BuildFiles and then the product's ups directory (that's what the trailing : means); that's why --build : means to only search ups directories[12]. The rsync should be clear, and the second eups distrib does the install.

What if your product isn't (yet) declared to ExtUPS? Write a build file scipy.build and maybe table file scipy.table in ~ACT/BuildFiles, and incant:

```
eups distrib -z act create -s /u/act/products/packages --build ~/ACT/BuildFiles: -S noeups=True scipy

eups distrib -z act install scipy 0.6.0 --root http://www.astro.princeton.edu/~act/packages scipy 0.6.
```

N.b. this uses scipy.table from the build file directory, ~ACT/BuildFiles. The noeups only applies at the top level; if your table file specifies dependencies they'll be honoured (the current versions will be used if you don't specify desired versions). A simplified version (in this case installing the just-released swig 1.3.35 on my laptop) is:

```
eups distrib create -s /u/act/products/packages --build \~/ACT/BuildFiles swig 1.3.35 -S noeups=True
eups distrib install swig 1.3.35
```

You can also use -S noeups=True to create a distribution from a checked out version without first installing it. You'll need to tell ExtUPS where to look for the table/build files in ./ups:

```
eups distrib create -s /u/act/products/packages --build ups -S noeups=True mwi 2.4
```

[12] You can use an empty string instead, but don't forget to quote it so that it isn't stripped by your shell, e.g. --build "''" — --build : is easier.

B.3.5 eups expandbuild

```
eups expandbuild -h
Usage:
    eups [commonOptions] expandbuild [options] buildFile [outDir]

    Expand a .build file as part of installing it.

    If outDir is provided, the expanded file will be written there;
    otherwise it'll be written to stdout unless you specify --inplace.

Options:
   -c, --cvs        arg    Use this CVSROOT
                           Alias: --cvsroot
   -i, --inplace           Modify file in situ
   -p, --product    arg    The name of the product
   -s, --svn        arg    Use this SVNROOT
                           Alias: --svnroot

Expand a .build file as part of installing it.

If outDir is provided, the expanded file will be written there;
otherwise it'll be written to stdout unless you specify --inplace.
```

Modify a ups build file (see section F.2) expanding certain variables:

@CVSROOT@ The value of `$CVSROOT` used to check out this product. This is guessed from `./CVS` if not specified with `--cvs` (or `-c`) or `$CVSROOT`.

@PRODUCT@ The name of the product. This is guessed from `buildFile` (by dropping the suffix `.build`) if not provided with `--product` (or `-p`).

@SVNROOT@ The `svn` locator (e.g. `svn+ssh://svn.lsstcorp.org/DC2/fw/trunk`) used to check out this product. This is guessed from `./.svn` if not specified with `--svn` (or `-s`) or `$SVNROOT`.

@VERSION@ The version of the product. This is taken from the `--version` (or `-V`) option.

E.g. the file `prod1.build`

```
export CVSROOT="@CVSROOT@"
cvs co -d @PRODUCT@-@VERSION@ -r @VERSION@ @PRODUCT@
```

becomes (with the `-V v1_0_1` option):

```
export CVSROOT="jeeves.astro.princeton.edu:/usr/local/cvsroot"
cvs co -d prod1-v1_0_1 -r v1_0_1 prod1
```

Because this list of pre-defined variables is so inflexible, you can define a dictionary of your own values in a startup file (*e.g.* `$HOME/.eups/startup.py`). For example, after

```
hooks.config.distrib["builder"]["variables"]["WHO"] = "RHL"
```

I can refer to `@WHO@` in build files.

A little imagination soon comes up with definitions such as:

```
hooks.config.distrib["builder"]["variables"]["DEFINE GET-SRC"] = """
get_src () {
    svn co "$@"
}
"""
hooks.config.distrib["builder"]["variables"]["GET-SRC"] = "get_src"
```

which allow the insertion of centrally controlled shell functions into all build files as they are expanded.

If a directory is specified, the modified build file will be written there, with the same name as the original; otherwise it's written to standard out unless you specify `--inplace`.

For example, the make target in a ups directory might contain the line:

```
eups expandbuild -V $(VERSION) ups/prod1.build $(prefix)/ups
```

There is one special case when rewriting build files: If the version is of the form `svn###` (e.g. `svn666`), any occurrence of `/tags/svn###` is replaced by `/trunk -r ###`. This will usually lead to `svn co svn+ssh://host.edu/product/` being replaced by `svn co svn+ssh://host.edu/product/trunk -r 666` which is probably what you want.

B.3.6 `eups expandtable`

```
Usage: eups expandtable [options] file.table [directory]
Options are
    -h, --help    Print this message
    -i, --inplace Modify file in situ
    -p, --product PRODUCT=VERSION   Expand PRODUCT to use VERSION
    -w, --warn    Warn about versions that don't start with v or V
```

Modify a ups table file replacing setupRequired and setupOptional lines which refer to the current version by the actual version number of the currently setup product; e.g.

```
setupRequired("-f ${EUPS_FLAVOR} astroda")
```

becomes

```
setupRequired("-f ${EUPS_FLAVOR} astroda v13_1")
```

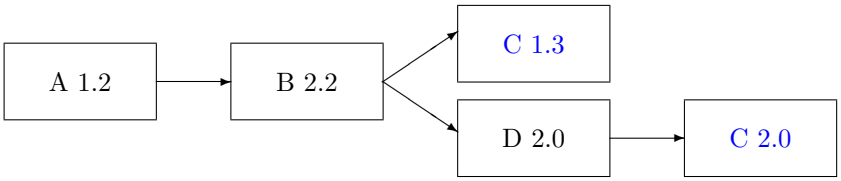
You can override the version with e.g. `-p astroda=rhl`; more than one `-p` command is permitted. If the table file contains a constraint on the version (e.g. `setupRequired(daf_base >= 3.2.6)`) it'll be preserved in the installed file (e.g. `setupRequired(daf_base 3.2.9 [>= 3.2.6])`). These options are used by the `VRO`'s `version` and `versionExpr` entries.

If a directory is specified, the modified table file will be written there, with the same name as the original; otherwise it's written to standard output unless you specify `--inplace`, in which case the substitution will be done in situ. You may omit file.table, or specify it as "-", to read standard input; this implies `--inplace`.

For example, the make target in a ups directory might contain the line:

```
eups expandtable -w iop.table $(IOP_DIR)/ups
```

In addition to adding explicit versions of all products listed in the table files, `expandtable` also writes a logical block to the tablefile that gives the versions of every product that it setup. Returning to the diamond dependency example in Sec. 4.1:



Let us assume that we chose to use C 2.0 (e.g. by explicitly setting up C 2.0 after setting up A). Initially A's table file looked like

```
setupRequired(B)
```

and `eups expandtable` rewrites it as

```
if (type == exact) {
   setupRequired(B   -j 2.2)
   setupRequired(C   -j 2.0)
   setupRequired(D   -j 2.2)
} else {
   setupRequired(B 2.2 [>= 2.2])
}
```

B.3.7 eups flags

```
Usage:
    eups [commonOptions] flags [options]

Print the value of EUPS_FLAGS
```

Print the value of EUPS_FLAGS

B.3.8 eups flavor

```
Usage:
   eups flavor
```

Print your machine/operating system's flavor. In simple cases, this'll be the same as `uname -m`, but in reality it has to worry about things like 32/64 bit libraries and os/x running on both PPC and X86 architectures.

B.3.9 eups list

```
Usage:
    eups [commonOptions] list [options] [product] [version]

Options:
   -c, --current           Only show current products
   -Z, --database   arg    Use this products path. Default: $EUPS_PATH
                           Alias: --with-eups
       --debug      arg    Permitted Values: raise
   -D, --dependencies      Print product's dependencies
       --depth      arg    Only list this many layers of dependency
   -d, --directory         Print product directory
   -e, --exact             Use the as-installed version, not the conditional in the table file
   -f, --flavor     arg    Use this flavor. Default: $EUPS_FLAVOR or `eups flavor`
   -F, --force             Force requested behaviour
   -h, --help              Print this help message
   -n, --noaction          Don't actually do anything
   -z, --select-db  arg    Select the product paths which contain this directory.
                           Default: all
   -s, --setup             Only show setup products
   -m, --table             Print name of table file
   -t, --tag        arg    List versions with this tag (e.g. current or stable)
   -T, --type       arg    Specify type of setup (permitted values: build)
   -v, --verbose           Be chattier (repeat for even more chat)
   -V, --version           Print the version
```

Print information about products.

The `--depth` option only applies to `--dependencies`. In general it's an expression; the useful ones are of the form `depth == 2` or `depth <= 2`. You may omit the `depth`, and if you just say `2` it's taken to mean `<= 2` and thus `depth <= 2`. E.g. the first-level dependencies may be displayed as `eups list --dependencies --depth ==1 product version`. Indentation's used to indicate the indirectness of the dependency unless you specify an exact depth (i.e. with `==`).

The command `setup list --current` will list all products that are declared current. Adding the `--verbose` option will also list the database root and product root paths. If you only want the product's directory, you can use `--directory`. The product name may be a glob expression (i.e. `*`, `?`, and `[]` are special, just like the shell). Note that you may need to quote them to protect them from the shell; e.g. `eups list \* svn\*` This command is deprecated; use `eups list` instead.

B.3.10 `eups path`

```
Usage: eups path [-h|--help] [n]

Print the product stack directories given via EUPS_PATH.  An
optional integer argument, n, will cause just the n-th directory
to be listed (where 0 is the first element).
```

Tell me the current `ExtUPS` search path

B.3.11 `eups pkg-config`

```
Usage:
    eups [commonOptions] pkg-config [options] product [version]

Print information about products

Options:
   -c, --cflags            Output all pre-processor and compiler flags
   -l, --libs              Output all linker flags
```

The same as the standard `pkg-config` command, except that it searches `$PRODUCT_DIR/etc` for the `.pc` file. It also works around the problem that quoted variables `$${...}` in `.pc` files appear as `%{...}` in `pkg-config` output.[13]

B.3.12 `eups remove`

```
Usage:
    eups [commonOptions] remove [options] [product] [version]

Remove a product

Options:
   -i, --interactive       Prompt user before actually removing products (default if -R)
   -N, --noCheck           Don't check whether recursively removed products are needed
       --noInteractive     Don't prompt user before actually removing products
   -R, --recursive         Recursively also remove everything that this product depends on
```

Undeclare the specified product, and remove it from the system. If you specify `--recursive` then all dependent products will also be removed; this is dangerous as they may be needed by someone else. `ExtUPS` accordingly checks for products that are in use and refuses to remove them unless you specify `--force`. As

[13] Yes; I filed a Debian bug report (378570) and it's fixed in the next release, 0.21-1

an added safeguard, `eups remove --recursive` will prompt you before doing anything (you can bypass this check with `--noInteractive`.

You can use `eups uses` to check whether a product's in use.

B.3.13 `eups startup`

```
Usage: eups startup [-h|--help]

List the startup files that customize EUPS (including $EUPS_STARTUP).  With
-v, show non-existent files that would be loaded [in brackets].
```

`ExtUPS` can be customised using files in a surprisingly large number of places; `eups startup` lists the startup files that are actually used (with `-v` non-existent files that would be used are also listed, enclosed in square brackets).

B.3.14 `eups tags`

```
Usage: eups tags [-h|--help] [options]

Print information about known tags, or clone or delete a tag

  --clone=CLONE         Specify a tag to clone (must also specify new tag)
  --delete=DELETE       Specify a tag to delete
```

For example,

```
$ eups tags --clone rhl tmp
```

assigns the tag `tmp` to all products tagged `rhl` (remember to declare `tmp` in your `startup.py` file with `hooks.config.Eups.userTags.append("tmp")`) while

```
$ eups tags --delete tmp
```

removes the tag `tmp` from all products (but doesn't undeclare it).

B.3.15 `eups undeclare`

```
Usage:
    eups [commonOptions] undeclare [options] product [version]

Undeclare a product

Options:
   -c, --current           Stop version from being current
   -t, --tag        arg    Undeclare versions with this tag (e.g. current or stable)
```

Undeclares `version` of `product` from the database. Also removes any tags attached to version; if you just want to delete a tag, use `--tag XXX` (as a special case rooted in history, `--current` is equivalent to `-t current`).

If you specify a tag but not a version then the version itself will *not* be undeclared; the exception is that it *will* be undeclared if version is of the form `tag:XXX`[14]

[14] *i.e.* the version name was automatically generated by a command such as `eups declare -t XXX -r .` or `eups declare -t XXX productName` where you didn't name a version

B.3.16 `eups uses`

```
Usage:
    eups [commonOptions] uses [options] product [version]

Show which products setup the specified product (and maybe version of that product)

Options:
   -c, --current          Look for products that setup product only because it's current
                          N.b. This isn't the same as specifying product's current version
   -d, --depth      arg   Only search down this many layers of dependency
   -e, --exact            Use the as-installed version, not the conditional in the table file
   -o, --optional         Show optional setups
```

Go through the `ExtUPS` databases and return every product that depends upon the specified product and version; if the version's omitted then everything that depends on product is returned, along with the version required. *N.b. this will be quite slow if you've disabled the cache — it'll need to read every table file that* ***`ExtUPS`*** *knows about.*

Unless you specify `--quiet` you'll be warned if a product depends on more than one version; this may sound unlikely but can happen if you setup the something more than once when setting up a product.

B.3.17 `eups vro`

```
Usage: eups vro [-h|--help] [options] product [version]

Print information about the Version Resolution Order (VRO) to use if issuing
the setup command with the same arguments.

Options:
  --type=SETUPTYPE      the setup type to use (e.g. exact)
  -c, --current         same as --tag=current
  -e, --exact           Consider the as-installed versions, not the
                        dependencies in the table file
  -r PRODUCTDIR, --root=PRODUCTDIR
                        root directory where product is installed
  -t TAG, --tag=TAG     Set the VRO based on this tag name
```

Sometimes you may want to know what VRO would be used by a particular `setup` command; here's how you can find out.

B.3.18 setup

```
Usage: setup [-h|--help|-V|--version] [options] [product [version]]

(Un)Setup an EUPS-managed product.  This will "load" (or "unload") the
product and all its dependencies into the environment so that it can be used.

Options:
  -C, --current         deprecated (use --tag=current)
  -Z PATH, --database=PATH
                        The colon-separated list of product stacks (databases)
                        to use. Default: $EUPS_PATH
  --debug=DEBUG         turn on specified debugging behaviors (allowed: raise)
  -e, --exact           Don't use exact matching even though an explicit
                        version is specified
  -f FLAVOR, --flavor=FLAVOR
                        Assume this target platform flavor (e.g. 'Linux')
  -E, --inexact         Don't use exact matching even though an explicit
                        version is specified
  -F, --force           Force requested behaviour
  -h, --help            show command-line help and exit
  -i, --ignore-versions
                        Ignore any explicit versions in table files
  -j, --just            Just setup product, no dependencies (equivalent to
                        --max-depth 0)
  -k, --keep            Keep any products already setup (regardless of their
                        versions)
  -l, --list            deprecated (use 'eups list')
  -m TABLEFILE, --table=TABLEFILE
                        Use this table file
  -S MAX_DEPTH, --max-depth=MAX_DEPTH
                        Only show this many levels of dependencies (use with
                        -v)
  -n, --noaction        Don't actually do anything (for debugging purposes)
  -N, --nolocks         Disable locking of eups's internal files
  -q, --quiet           Suppress messages to user (overrides -v)
  -r PRODUCTDIR, --root=PRODUCTDIR
                        root directory of requested product
  -z DIR, --select-db=DIR
                        Select the product paths which contain this directory.
                        Default: all in path
  -t TAG, --tag=TAG     Put TAG near the start of the VRO (may be repeated;
                        precedence is left-to-right)
  -T TAG, --postTag=TAG Put TAG after version(Expr)? in VRO
                        (may be repeated; precedence is left-to-right)
  --type=SETUPTYPE      The setup type to use (e.g. exact)
  -u, --unsetup         Unsetup the specifed product
  -v, --verbose         Print extra messages about progress (repeat for ever
                        more chat)
  -V, --version         Print eups version number
  --vro=LIST            Set the Version Resolution Order
```

This is the workhorse routine. It sets up `version` of `product`. You may omit the `version` (in which case

you'll get the `current` version), specify an explicit version, or use a logical expression. E.g.

```
setup ds9
setup ds9 4.13
setup ds9 '>= 5.0'
setup ds9 '>= 5.0 || rhl'
```

(don't forget to quote `<` and `>` as they're interpreted by the shell).

Important: Since the commands to set environment variables differ across shells, it is essential that `$SHELL` reflects the current shell.

Options `--current` and `--setup` are only relevant with `--directory`.

(N.b. setup is really an alias or shell function that runs the command
`source 'eups_setup [options] product [version]'`
`eups_setup` writes a shell script that is then sourced into the current shell. This `eups_` command alone is not deprecated.)

If you specify `--root`, you may not need to give the product's name if you're not giving a version number too. If the ups directory contains only one table file, it'll be taken to be the product that you want to setup. I.e. `setup -r .` will work.

If you set verbose to three or more, the file that `setup` writes (and which is sourced to modify your current shell's variables) is not deleted after use, allowing you to peruse it at your leisure.

The `-k` ('keep') makes `ExtUPS` preserve your pre-existing setups. You can achieve the same effect by carefully chosing your setup order, but this is easier.

The `-D` option specifies that the product itself *not* be setup, but that it's dependencies should be processed as normal.

With `-M` you specify a table file for something that may not even be declared to `ExtUPS`; this is useful for setting up dependent products during installation. Implies `-D`, and you may not specify a product name. *N.b.* this is not the way to set up a local product — use `setup --root dir [product]` (e.g. setup -r .).

If `ExtUPS` detects an error, the file-to-source will be empty (i.e. the setup will be aborted) unless you specify `--force`.

Usually `ExtUPS` will lock files before using them, but this may not be desireable or required when setting up products on a quiescent system (i.e. one where no-one is actively declaring products). You can turn locking off with the `-N` option. Another way to achieve this is to use the `commandCallbacks` mechanism; there's an example in Sec. B.2.2.

B.3.19 `unsetup`

```
Usage:
   unsetup [options] product
Options:
(same as setup)
```

All the notes under `setup` apply to `unsetup`. Unsetting up a product relies on the environment variable `$SETUP_<product>`, so it fails if the variable isn't set (unless you use `-M`).

B.4 Environment Variables

Required variables -

- `SHELL` This must be set to the shell that you're running. This is generally set correctly for you, unless you run a shell script, or change your shell.
- `EUPS_DIR` Where `ExtUPS` is installed.

- `EUPS_PATH` The colon-separated path of root directories under which all products are installed. Each root directory has a UPS database in a top-level `ups_db` directory. As of `v0_7_33`, when you source `setups.[c]sh`, any pre-existing value of `EUPS_PATH` is preserved at the end of your new path.

Optional Variables -

- `EUPS_DEBUG` This sets the verbosity; equal to the difference between the number of `-v`s and `-q`s.

  If you are confused by some error, or simply nosy, try adding a `-v`.

- `EUPS_FLAVOR` Set to the default flavor (if not set `ExtUPS` will use the value returned by `eups flavor`).

- `EUPS_FLAGS` A list of options prepended to all setup/unsetup/declare/undeclare commands. Options that are not permitted by a given command are silently ignored (e.g. `-k` with `eups declare`).

B.5 Locking

There are currently has three types of locks, and each command chooses one of them.

- no lock
- shared lock
- exclusive lock

and the entire `ExtUPS` system is locked[15] Locking is per-process, or more precisely per a process and its children.

The following commands take a shared lock:

- `eups admin info`
- `eups admin listCache`
- `eups distrib list`
- `eups expandbuild`
- `eups expandtable`
- `eups list`
- `eups pkg-config`
- `eups tags`
- `eups uses`

and these take an exclusive lock:

- `eups admin buildCache`
- `eups admin clearCache`
- `eups admin clearServerCache`
- `eups declare`
- `eups distrib clean`

[15] This may be a problem...

- `eups distrib create`
- `eups distrib declare`
- `eups distrib install`
- `eups remove`
- `eups undeclare`

B.6 The Database

B.6.1 `ExtUPS` Files

This section briefly describes the structure of the UPS database. Each EUPS root directory in `EUPS_PATH` contains a database directory `ups_db`, with a subdirectory for each declared product. Each of these product directories contain a series of product definitions, specified by three types of files:

- current.chain

  There is only one (if any) of these per directory. The current.chain file specifies the current version for a given flavor, and is what is used as the default if the version is not specified.

- Version files

  There is one of these for every declared version of the product, named `<version>.version`. The file specifies basic information about the version of the product, including where it is installed.

- Table files

  These specify the dependencies of the product. The default name for these is `<version>.table` although a specific name can be specified in the version file.

  Table files use a series of commands to specify the dependencies. These are discussed in the next subsection.

  NOTE: See Section 5 for a note on the search algorithm that `ExtUPS` uses for locating table files.

- Cache files

  `ExtUPS` uses a set of cache files that record the contents of the declared table files. This provides a *very* significant speedup when using commands such as `eups remove` or `eups uses`, but the speedup is also significant with a lowly `eups list`.

  The cache can get confused; in this case you can use the `eups admin clearCache` command to clean up. If you can figure out exactly what you did to confuse `ExtUPS` please let us know so we can fix it! There is also a problem where the cache grows large; once more `eups admin clearCache` should solve (or rather, mask) the problem.

  We are thinking of providing an option to disable the cache, either because you don't have write permission, or because the risk of cache corruption is more important to you than speed considerations.

B.6.2 `ExtUPS` table format

Table files consist of blocks of commands, possible subject to if-else if-else blocks, *e.g.*

```
if (FLAVOR == DarwinX86) {
   print(stdinfo, os/x)
} else if(FLAVOR == Linux || FLAVOR == Linux64) {
   print(stdinfo, linux)
}
```

By default the test may use `FLAVOR` or `TYPE` (*e.g.* `setup --type build`) (you can add more by setting `config.Eups.setupTypes` in a `startup.py` file; see B.2.1)

ExtUPS table commands If a value looks like $?{NAME}, and there is no such environment variable, the entire line is ignored (As a special case, PRODUCT_DIR is considered undefined if its value is none). If you omit the ? it's an error for NAME not to exist. Additionally, ${NAME:-XXX} expands to $NAME if defined, and to XXX otherwise.

The following table commands are supported by ExtUPS: (<Name> either specifies an alias or an environment variable - clear from the context.)

- addAlias(<Name>,<Value>) Defines a function or an alias. Arguments should be written in sh format (i.e. \"$@\" not \!*); don't worry, the csh alias will be defined correctly. E.g.

  ```
  addAlias(foo, source `${PRODUCT_DIR}/bin/eups_setup setup \"$@\"`;);
  ```

  Don't forget to escape the quotes.

- envAppend(<Name>, <Value> [,<delimiter>]) Appends <Value> to <Name>. The default delimiter is ':'; the value may not contain an embedded delimiter (use two separate envAppend commands instead). If the value looks like $?{name}, and there is no such environment variable, the envAppend command is ignored. If <Value> starts or ends with <Delimiter>, the final value of <Name> will have a delimiter pre- or -appended; this is useful for variables such as $MANPATH which use such an empty element to mean, "use the default path too". E.g.

  ```
  envAppend(MANPATH, :${PRODUCT_DIR}/man)
  ```

- envPrepend(<Name>,<Value> [,<delimiter>]) Similar to envAppend, but put the value at the beginning

- envSet(<Name>, <Value>) Set <Name> to <Value>.

- envUnset(<Name>) It might sound as if this directive unsets the environment variable <Name>, and it would be possible to implement this if you have a use for it. However, at present the only supported <envUnset> command is envUnset(PRODUCT_DIR) which may be used to prevent ExtUPS from setting $PRODUCT_DIR. For example, git uses GIT_DIR for its own purposes, but yet it's useful to use ExtUPS to manage git versions.

- print(<String>)

- print(<Dest>, <String>) Print the string to stdout; if <Dest> is provided it must be one of stdout, stderr, stdok, stdinfo, and stdwarn.

- setupRequired(<Name> [options] [<Version>]) Setup <Name>. If no <Version> is specified, the current version is used. Fails if unable to setup.

  Version may in fact be a logical expression, specifying acceptable versions. Permitted relational expressions are >, >=, ==, <, and <=. More than one expression may be joined using || (&& is not supported due to laziness). For example,

  ```
  setupRequired(foo 1.2)
  setupRequired(foo == 1.2)
  setupRequired(foo >= 1.2)
  setupRequired(foo < 1.3)
  setupRequired(foo >= 1.2 || == rhl)
  setupRequired(foo >= 1.2 || rhl || svn)
  ```

The == is optional. See the Sec. B.6.3 for the definition of the ordering applied to version names.

If more than one version matches the specified condition, priority is given to the version declared current; if there is no current version, or it doesn't satisfy the condition, the highest valid version number is used.

You may sometimes want to ignore explicit versions (e.g. if you are testing a new version, installed and current as `rhl`, but the table file specifies a version `> 1.2`). You can do this with `setup -i` or `--ignore-versions`.

- `setupOptional(<Name> [ <Version>]` Same as `setupRequired` except that it does not fail if unable to setup.
- `setupRequired(<Name> [options])` Unsetup `<Name>`. Fails if unable to unsetup.

  An example is the LSST processing stack, where setup algorithms are run and a top-level product sets up a default set. If want to disable some it's convenient to unsetup them up in your own product:

```
setupRequired(pipe_tasks)
unsetupRequired(meas_extensions_multiShapelet -j)
print(stdinfo, Disabling model fitting)
```

- `unsetupOptional(<Name> [options]` Same as `setupRequired` except that it does not fail if unable to unsetup.

B.6.3 Version ordering

Version names are taken to be of the form `PrefixAAA.BBB.CCC-LLL.MMM.NNN+XXX.YYY.ZZZ` where `Prefix` is optional, as are `-LLL.MMM.NNN` and `+XXX.YYY.ZZZ`. Let's call `PrefixAAA.BBB.CCC` the 'primary' part, `LLL.MMM.NNN` the 'secondary' part, and `XXX.YYY.ZZZ` the 'tertiary' part. Each of the 'dotted' parts may have any number of components. If you wish, you may use _ instead of . as separators (this makes cvs happier). Examples of valid versions are:

```
1.2.3
v1.2.3.4
a.b.c
1.2.3-a.b
1.2.3+a.b
1.2.3-rc2+a.b
```

The sorting rules are:

1. If the version starts with non-digits (i.e. `Prefix` is provided), the two versions must have identical prefixes, or the first will sort to the left of the second (i.e. it can never satisfy <, <=, or ==).

   If `Prefix` matches, it's ignored for the rest of the comparison.

2. The primary part is then split on . (or _), and each pair of components in the two names is compared numerically (if they look like integers) or lexicographically (otherwise). If they differ, the version name with the "smaller" component is taken to sort to the left, and the comparison stops.

3. If one of the two version names is a subset of the other the longer version sorts to the right. This only applies to the primary part; any secondary part is ignored in this comparison.

4. If the two primary parts are identical, then the secondary part is examined. Apart from the fact that no `Prefix` is allowed, the same rules are applied. If only one version has a secondary part, it sorts to the *left* – so `1.2-rc2` sorts to the left of `1.2`.

5. If the primary and secondary parts are identical, then the tertiary part is examined, applying the same rules as the secondary, except that if only one version has a tertiary part it sorts to the *right*, so `1.2+hack` sorts to the right of `1.2`.

What's the point of these complicated rules? The primary sort should be intuitive. People often name *pre*-release candidates things like `1.2-rc2`, hence the rule for `1.2` being later than `1.2-rc2`. After a version is released, you sometimes need a fixed version, so its name should sort later — hence `1.2+bugfix`.

Here are some examples:

V1	V2	compar	notes
aa	aa	0	
aa.2	aa.1	+1	Sort components numerically
aa.2.1	aa.2	+1	
aa.2.1	aa.2.2	-1	
aa.2.1	aa.3	-1	
aa.2.b	aa.2.a	+1	Sort components lexicographically
aa.2.b	aa.2.c	-1	
v1_0_3	a1.0.2	-1	Mismatching prefixes always sort to the left
v1_0_0	1.0.2	-1	Only V1 has a prefix
1_0_0	v1.0.2	-1	Only V2 has a prefix
v1_0_2	v1.0.0	+1	You can mix `.` and `_`
v1.2.3	v1.2.3-a	+1	primary parts are identical, so secondary is examined
v1.2-0	v1.2.3	-1	primary parts differ, so secondary parts are ignored
v1.2-4	v1.2.3	-1	primary parts differ, so secondary parts are ignored
1-rc2+a	1-rc2	+1	primary and secondary match; tertiary sorts to right
1-rc2+a	1-rc2+b	-1	sort is on tertiary

(Here compar is the value that C's `qsort` would use; -1 if V1 is less than V2; 0 if they're equal; +1 if V1 > V2).

B.6.4 Obsolete table commands

- `pathAppend`, `pathPrepend`, `pathRemove`, `pathSet` Aliases for the `env` commands.
- `prodDir()` Sets `<PRODUCT>_DIR` to the directory where the product is installed; this directive is ignored as `<PRODUCT>_DIR` is automatically set for you.
- `setupEnv()` Sets `SETUP_<PRODUCT>` so that product can be unsetup. this directive is ignored as `<PRODUCT>_DIR` is automatically set for you.

B.6.5 Variables

The arguments to these UPS commands can be variables. All environment variables can be used as `${<ENV_VARIABLE>}`; *e.g.* `${HOME}`. `$HOME` will not be expanded, so `envSet(FOO, ${HOME})` set `FOO` to the *value* of `$HOME` (*e.g.* `/u/rhl`), whereas `envSet(FOO, $HOME)` sets it to `$HOME`.

In addition, the following special variables are defined;

- `${PRODUCTS}` The database
- `${PRODUCT_DIR}` Where the product is installed.
- `${PRODUCT_DIR_EXTRA}` Where the product's extra files are installed; only set if there actually are any extra files (as provided by `eups declare`'s `-L` and `-M` options; B.3.3)
- `${PRODUCT_FLAVOR}` The product flavor.
- `${PRODUCT_NAME}` The product name

- ${PRODUCT_VERSION} The product version
- ${UPS_DIR} The ups directory of the product (distinct from the directory in ${PRODUCTS} - this is often where the table file will be stored).

B.7 ExtUPS autoconf commands

XXX Finish me

- AC_INIT(product, version, ... This is a standard autoconf line, but with an invalid version name (X.Y).
- AC_DEFINE_ROOT(version [, flavor]) Define the configure options
  - --with-eups=path Specify the value of EUPS_PATH and the prefix (defaults: $EUPS_PATH if set, or $prefix/share. The prefix is taken to be the first component of the prefix, unless --with-eups-db is specified.
  - --with-eups-db=DB Select which component of the path is the prefix (just like setup's -z). For example, if EUPS_PATH is /u/dss/products:/u/act/products then the default prefix is /u/dss/products, but with -with-eups-db=act the prefix becomes /u/act/products.
  - --with-flavor=XXX Set the value of EUPS_FLAVOR (default: $EUPS_FLAVOR, or the value returned by eups flavor or, failing **that**, uname).

  The conventional files are:
- etc/builddefs Supports building on this platform. E.g.

```
CC = cc -std=c99
CFLAGS = -g -Wall
RANLIB = ranlib
```

- etc/installdefs Supports the make install and make declare targets.
- etc/makedefs Supports building and linking products (GSL in this case). After expansion, the file will look like:

```
GSL_CFLAGS = -I$(GSL_DIR)/include
GSL_LIBS = -L$(GSL_DIR)/lib -lgsl -lgslcblas -lm
```

C Installing ExtUPS

If you want to play with ExtUPS you may want to install a copy.

You can check out a copy from svn from NCSA:

```
svn co svn://svn.lsstcorp.org/eups/trunk eups
```

or to get an released version e.g.

```
svn co svn://svn.lsstcorp.org/eups/tags/1.2.6 eups
```

You can of course list available versions with

```
svn ls svn://svn.lsstcorp.org/eups/tags
```

Alternatively, grab the tarball

```
http://www.astro.princeton.edu/~act/packages/eups.tar.gz
```

which is probably out of date.

Then unpack it, configure, and install. E.g.

```
zcat eups.tar.gz | tar -xf -
cd eups
./configure --with-eups=/u/act/products
make show
make install
```

The `make install` command will tell you to source a file with a name such as

```
/u/act/products/eups/bin/setups.sh
```

which will make `ExtUPS` commands available to you; csh users should use `setups.csh`. You'll probably want to add this line to your shell startup file (.bashrc or .tcshrc).

C.1 Details, details...

You can use `make show` to show you where things are going to be installed:

```
$ make show
You will be installing ups in $EUPS_DIR  =  /usr/local/products/eups
Eups will look for products in $EUPS_PATH = /usr/local/products
Your EUPS database[s] will be               /usr/local/products/ups_db
Your EUPS version is                        1.2.6
Your site configuration files will be in    /usr/local/products/site
```

If you already have the environment variable `$EUPS_PATH` set, then `EUPS_PATH` will default to the old value.

You can control where files are put with the following configure options:

```
--prefix=PREFIX         install architecture-independent files in PREFIX
                        [default: /usr/local]
--with-eups=PATH        Use PATH as root for installed products
--with-eups-db=DB       Select component of path to use for EUPS_DIR;
                        "/eups" will be appended [default: first]
--with-eups_dir=DIR     Install eups into DIR/{bin,doc}
--with-setup-aliases=name1:name2  Make name1 an alias for setup, and
                        name2 an alias for unsetup
```

The difference between `--prefix` and `--with-eups` is that the former appends `/share` to the specified directory, to be consistent with the standard default value, `/usr/local`.

If you don't explicitly choose a `EUPS_DIR` with `--with-eups_dir` or `--with-eups-db` the value from the environment will be adopted. If you don't have `EUPS_DIR` set, the first component of the `PATH` will be used, with `/eups` appended.

As an alternative to

```
./configure --with-eups=/foo/bar/share
...
make install
```

you can say

```
make install EUPS_PATH=/foo/bar/share
```

`make install prefix=...` also works as an alternative to `--prefix=...`. Note that `prefix` is ignored if you've set `EUPS_PATH` either explicitly or implicitly via an environment variable; `make show` is your friend.

The `--with-setup-aliases` option is provided for those who find the commands 'setup' and 'unsetup' inelegant, or are who are concerned about the conflict with the command `/usr/bin/setup` on many Linux systems. A typical usage would be

```
configure --with-setup-aliases=activate:deactivate
```

C.1.1 Working with multiple copies of ExtUPS

This may only be of interest to the authors. You can declare ExtUPS as a full-up ExtUPS product in its own right, and use `setup eups version` to switch between them. To make this easier, the main Makefile allows you to say, `make declare` to declare your new version.

D Changes

D.1 New features

ExtUPS has undergone significant enhancements recently. Some highlights are:

- The addition of $EUPS_PATH
- The ability to specify logical expressions instead of simple version numbers in requirements
- Support for adding an extra delimiter at the start/end of a path variable (e.g. MANPATH)
- The ability to use relative paths with `--root`
- The ability to omit the product name when you're setting up with `--root`; e.g. `setup -r .`
- `eups distrib` can now use "build files" to build products locally; `sh` scripts that play nicely with ExtUPS.
- `eups distrib` can now use scp as well as http to retrieve files.
- support `$?{name}` in table files
- Don't print setup/current with `eups list -d`
- Support `eups declare -r XXX version`
- Allow version expressions in `eups list`
- Support `eups path n` to print part of a EUPS_PATH
- Added `setup -C` as an alias for `setup -c`, for consistency with `eups distrib`
- Added `eups distrib -S noeups=True` to generate build/manifests for non-eups or non-installed products
- Allow `eups declare` to guess the directory given the product and version, e.g. `eups declare moby 3.1.4`.
- Don't allow a user to undeclare a setup product (as it couldn't be unsetup later)
- Support `eups declare -n`
- Support `~` (but not `~user`) when using -r to specify product directories; this only matters in table files (where the shell didn't already expand the ~ for you)
- Support `eups list prod LOCAL:/product/dir`
- Support `setup --max-depth/-S` for use with `setup -v`.
- Add `eups distrib list`
- Add `eups undeclare --force` to undeclare products that are currently setup
- Add `eups remove` to remove installed products
- `eups uses` to list everything that depends on a given product

D.2 Obsolete features

- The environment variable `$EUPS_PATH` used to be known as `$PROD_DIR_PREFIX`. This is no longer supported.
- The `ExtUPS` database used to be given by the environment variable `$PRODUCTS` with a default value of `$PROD_DIR_PREFIX/ups_db`. This is no longer supported.

E `ExtUPS` and autoconf/configure

If you're a user, you don't need to understand this section. If you're creating new products, you may need to.

The `ExtUPS` distribution includes a set of autoconf macros in the file `etc/ac_eups.m4`. Once you've arranged for autoconf to know about this file,[16] you can write a `configure.ac` file that looks something like

```
AC_INIT([myProduct], [X.Y], [rhl@astro.princeton.edu])
EUPS_DEFINE_ROOT($Name:  v2_4 $)
AC_PROG_CC(cc)
EUPS_WITH_CONFIGURE([gsl])
EUPS_WITHOUT_CONFIGURE([fftw3], [fftw3.h], -lfftw3f, [fftw3f,fftwf_plan_dft_2d])
AC_CONFIG_FILES([etc/Makefile.global])
AC_OUTPUT
```

[17]

Once you've generated a configure file (i.e. typed `autoconf`)[18] you can say

```
cd $HOME/myProduct
setup gsl; setup fftw3
./configure --with-eups=/u/act/products --with-flavor=Linux64
make
make install
make declare
setup myProduct v2_4
```

You might wonder how you were supposed to know that you had to type `setup gsl` and `setup fftw3` in order to build `myProduct`. The answer, is: "It depends". If you're `myProduct`'s loving author, you knew that you needed `gsl` as you included e.g. `gsl/gsl_rng.h` in your source. If you're not the author, then `myProduct` presumably includes a properly written table file (probably called `myProduct/ups/myProduct.table`) which knows about `gsl` and `fftw3`:

```
setupRequired("fftw3")
setupRequired("gsl")
```

in this case what you *really* typed was:

```
setup --root $HOME/myProduct myProduct
```

which setup `gsl` and `fftw3` for you:

```
Setting up:      fftw3              Flavor: Linux      Version: v3_0_1
Setting up:      gsl                Flavor: Linux      Version: v1_8
```

The `make declare` command told `ExtUPS` about your newly built product; the `setup myProduct v2_4` command told the system that you wanted to use it.

How did your `configure.ac` file achieve this magic?

[16] e.g. by copying the file into your top level directory

[17] The old forms with `UPS_` (e.g. `UPS_DEFINE_ROOT`) are supported for backward compatibility.

[18] You'll check the resulting configure file in, so users won't need to have autoconf on their machines

- AC_INIT

  This is a standard autoconf line, but with an invalid version name (X.Y).

- AC_DEFINE_ROOT

  Define the configure options --with-eups=path, --with-eups-db=DB, and --with-flavor=XXX. The version comes from the $Name: v2_4 $ — if the v2_4 were absent the value cvs would have been used instead.

- AC_PROG_CC(cc)

  Find the C compiler

- EUPS_WITH_CONFIGURE([gsl])

  Tell configure that we need the product gsl, and that it comes with a pkg-config script.[19] In this case, it came from ups, but the macro also supports --with-gsl=DIR; there are other possibilies (see the reference manual, Sec. B.7). If all goes well the macro macro sets (in the case of gsl) the variables GSL_CFLAGS and GSL_LIBS.

- EUPS_WITHOUT_CONFIGURE([fftw3], [fftw3.h], -lfftw3f, [fftw3f,fftwf_plan_dft_2d])

  This macro is similar to EUPS_WITH_CONFIGURE, and sets FFTW_CFLAGS and FFTW_LIBS, armed with knowledge of ExtUPS and --with-fftw3=DIR.

  The difference is that this time configure has to probe the system to find the include files and libraries. More specifically, it checks that it can find fftw3.h and that -lfftw3f provides the symbol fftwf_plan_dft_2d.

- AC_CONFIG_FILES

- AC_OUTPUT

  The standard autoconf macro to interpolate derived values into Makefile.global. What does this mean? **XXX** Describe Makefile.global.in

E.0.1 A Strategy for using autoconf and ExtUPS to Manage Layered Products

A more complete example is:

```
AC_INIT([myProduct], [X.Y], [rhl@astro.princeton.edu])

dnl --------- UPS directories and how to install --------------------------

EUPS_DEFINE_ROOT($Name:  $)
EUPS_INSTALL_DIRS

dnl --------- Build environment ------------------------------------------
AC_PROG_CC(cc)
AC_PROG_RANLIB
dnl --------- Products from UPS ------------------------------------------
dnl (remember to update etc/Makefile.global if you add a new product)

EUPS_WITH_CONFIGURE([gsl])
EUPS_WITH_CONFIGURE([pslib])

EUPS_WITHOUT_CONFIGURE([cfitsio], [fitsio.h],
-lcfitsio, [cfitsio,ffopen])
EUPS_WITHOUT_CONFIGURE([fftw3], [fftw3.h],
```

[19] I.e. a script gsl-config which accepts arguments --cflags and --libs to print out the options that you should pass to the compiler and linker to use the gsl libraries. For example, --libs prints -I/usr/local/include and --cflags prints -L/usr/local/lib -lgsl -lgslcblas -lm

```
-lfftw3f, [fftw3f,fftwf_plan_dft_2d])
EUPS_WITHOUT_CONFIGURE([wcslib], [wcslib/wcslib.h],
-lwcs, [wcs,wcsini])

AC_CONFIG_FILES([etc/makedefs etc/builddefs etc/installdefs])
AC_CONFIG_FILES([bin/eups_import], [chmod 755 bin/eups_import])

AC_OUTPUT
```

F "Transport Layers" for `eups distrib`

`pacman` is a package used by the HEP community (e.g. Atlas) to distribute software. It doesn't solve the same problem as `ExtUPS` (supporting multiple simultaneous versions of the same package), but it does nicely support building packages from source. The home page is `http://physics.bu.edu/pacman/`.

An example `pacman` script is

```
#
# Sample product installation using pacman
#
# author: RHL
# data:   2006/12/04
#
packageName('hello')
url('Foo', 'file:///dev/null')
#
# N.b. version is also encoded in directory name below
#
version('v1_0_0')
#
# Package requirements
#
platformGE("unix")
which("gcc"); which("make")
#
# Prepare to install
#
downloadUntar('/Users/rhl/LSST/Pacman-test/packages/hello-v1_0_0.tar.gz', 'HELLO_DIR')
cd('$HELLO_DIR')
cd('v1_0_0')
shell("make")
shell("make install")
```

F.1 Using `pacman` with `ExtUPS`

To use `pacman`, simply specify a `pacman` cache when running `eups distrib create`:

```
eups distrib -z lsst --pacman http://lsst.org/cache create hello
```

The `eups distrib install` command is unchanged:

```
eups distrib install -r /u/lsst/products/packages/Darwin -z act hello v1_0_0
```

F.2 Using 'Build Files' with `ExtUPS`

This isn't an integral part of eups, but it can be very useful in conjunction with `eups distrib`.

The work of installing a package frequently reduces to a stereotyped set of operations; for each dependency you type:

```
cvs co -r XXX prod1
cd prod1
make install
cd ..
```

followed by `eups declare ...` and on to the next product. It'd be nice not to have to type this repeatedly, especially if you have to install on a bunch of machines that don't share an `$EUPS_PATH`. Fortunately, you don't have to.

One approach would be to bite the bullet and learn `pacman` (see Appendix F); another is to write simple 'build files'. The former is far more sophisticated and bullet-proof; the latter is simpler. *Note: we may remove* ***pacman*** *support in a future* ***ExtUPS*** *release*

If you put a file `prod.build` in `prod/ups` and install it along with the table file, `eups distrib create --build :` will set things up so that a subsequent `eups distrib install` will copy the build file to the new machine and execute it; if you have a set of dependent products they will be executed in the proper order. By default, the products will be downloaded and built in `EupsBuildDir` in a flavor subdirectory of your `$EUPS_PATH`; e.g. `/u/act/products/DarwinX86/EupsBuildDir` (but you can change this with the `-T` flag). The build files are put in the same place, in case you want to look at them or fix buglets (simply say `sh prod-1.2.3.build`). If your verbosity is 3 or higher, the commands will be executed with `-v -x` set.

At their simplest, build files could simply be the commands that you'd type (such as those listed above); `ExtUPS` will take care of the declarations. This isn't very satisfactory, as it requires the build file to include the version string, and that the correct value of `$CVSROOT` be already set[20] Because `eups distrib` chains commands together with `&&` (to check that the previous command succeeded) comments in build files are slightly problematic (in particular ones at the top of your file, as `eups distrib` usually preceeds your commands with some of its own while preparing to do the distribution). Accordingly, lines starting with `#` are rewritten to use the shell's executable comment, `:` instead.

This is where the `eups expandbuild` command comes in; it's used to install build files expanding `CVSROOT` (or `SVNROOT`), `PRODUCT`, and `VERSION`. This allows you to write generic build files such as `prod1.build`

```
export CVSROOT="@CVSROOT@"

cvs co -d @PRODUCT@-@VERSION@ -r @VERSION@ @PRODUCT@
cd @PRODUCT@-@VERSION@

setup -r .
make install
```

If your `make install` target says something like

```
  eups expandbuild -V $(VERSION) ups/prod1.build $(prefix)/ups
```

the installed version (assuming that `$(VERSION)` is `v1_0_1`) becomes

```
export CVSROOT="jeeves.astro.princeton.edu:/usr/local/cvsroot"

cvs co -d prod1-v1_0_1 -r v1_0_1 prod1
cd prod1-v1_0_1

setup -r .
make install
```

[20] This is problematic if you're getting code from a number of different places.

and everyone's happy. [21]

We do try to check that build files are sensible (although you can skip the checks with `--force`; more specifically, we look for a `curl`, `cvs`, `svn`, or `wget` command. If you want to bypass this check, include a comment line containing the phrase `Build File` somewhere in your file, e.g.

```
# Tell eups distrib that this is a valid Build File
```

[21] There's a subtlety here. Imagine that we had a product `prod2` which depended on `prod1`. As we source each build file separately, the `setup -r .` in `prod1`'s build file isn't visible by the time that `prod2`'s being built (as it's run in a different shell), so `prod1` isn't setup. Even if it were, `prod2` tries to setup `prod1` itself (as it's a dependency), which requests the *current* version of `prod1`. We don't want `eups distrib` to always declare products current, so what should we do? The answer is that `eups distrib` inserts explicit setup statements into the the build script and replaces the `setup` with `setup -j`; try running with the `--verbose` flag if you want to see this in action, or simply read the installed build file.