Table of Contents

Table of Contents

RootCore

Comment: page moved from AtlasProtected to the Atlas web, other topics also will be moved.

Introduction

RootCore is a package that helps developers build packages that work standalone (outside Athena). It works both for packages that are ROOT-only and for packages that work with both Athena and ROOT. RootCore is meant to make life easier for all parties involved:

- the package developer has an easier life, because he no longer has to maintain his own makefile,
- the package user has an easier life, because packages have a standard build prescription and are more likely to build without problems and
- the support group has an easier life, because the build process is more standardized and problems can often be addressed centrally.

The basic creation of a RootCore environment is straightforward. RootCore depends on ROOT, so ensure that ROOT is accessible: the environment variable ROOTSYS should beset, and the ROOT bin directory should be in the PATH. Suppose you have a list of all the packages that you need in the file `packages.txt`, then you can set up your entire RootCore environment with the following lines:

```
mkdir packages
cd packages
svn co svn+ssh://svn.cern.ch/reps/atlasoff/PhysicsAnalysis/D3PDTools/RootCore/tags/`svn ls svn+ss
source RootCore/scripts/setup.sh
rc checkout packages.txt
rc find_packages
rc compile
```

That's it. If you want a little more control over what goes on, you will find descriptions of how to do each step that build does in the document below. Please note that you have to repeat the last command in each session you want to use RootCore and if you are using a c-shell you will have to source `setup.csh` instead. Also note that it is recommended to use the latest tag of RootCore (which the above may or may not be).

WARNING: for historical reasons all the scripts in RootCore have the suffix .sh. Except for the setup script, you should NOT source them. This will cause all kinds of errors and has no advantages. Just call them directly as in the instructions above.

If Something Goes Wrong

If something goes wrong, first try to clean and recompile everything. The following assumes that you sit inside a directory that contains your RootCore package itself as well as all the other packages you want to build. In the above example that would be the directory packages. Start out with a clean shell that has root and everything else (except for RootCore) setup already:

```
cd RootCore
source scripts/setup.sh
cd ..
rc find_packages
rc clean
rc compile
```

If that doesn't fix your problem, make sure that you are using either the latest tagged or the trunk version of RootCore. The problem you are having may already have been fixed. If you find that an older version works for you, and the newest does not, please report this as well. The latest version is always supposed to work in all circumstances. If you find that the prescriptions on this page do not work for you, but an alternate prescription does, report it as well. This page is the only official source of RootCore documentation, so if something is missing or wrong it needs to be addressed.

To report an error, send a mail to ALL of the following addresses. Note that you need to do this from a CERN email address or you won't be able to submit to CERN mailing lists. If you don't have a CERN email address, just send the message to me. Please make sure to include RootCore in the subject line. If you can please include the output of `rc version` or otherwise at least the RootCore version you are using. When reporting grid problems please include the panda version as well.

- Nils.Erik.Krumnack@cern.ch, myself as the package developer and maintainer.
- hn-atlas-PATHelp@cern.ch, the list of PAT developers. They are able to answer most questions in case I am unreachable.
- only for grid-related problems: hn-atlas-dist-analysis-help@cern.ch, the support list for user grid problems. While they are not experts on RootCore they can spot problems that are grid-related and not RootCore specific.

Individual Build Commands

While the prescription above should give you a jump start at setting up your RootCore installation, they are a little inflexible when it comes to your everyday need as a developer. Below you find the individual commands that are incorporated in the master `build.sh` script.

Setting up RootCore

To set up RootCore, first ensure that you have a working ROOT version set up and in your path. Next check out RootCore and set it up. This example dynamically sets up what is the latest RootCore tag at the moment. For RootCore this is considered safe, and it is recommended to use the latest tag:

```
svn co svn+ssh://svn.cern.ch/reps/atlasoff/PhysicsAnalysis/D3PDTools/RootCore/tags/`svn ls svn+ss
```

Now you have to source `setup.sh` or `setup.csh`, depending on which kind of shell you use. If you update RootCore, you should probably also run `rc clean` and `rc find_packages` to make sure you don't have any remnants from the previous version laying around.

If you have a machine with multiple cores (most machines these days), you can instruct RootCore to use more than one core through the environment variable ROOTCORE_NCPUS. If you set that to the number of CPUs, `rc compile` will be using that many parallel processes for compilation.

```
export ROOTCORE_NCPUS="$(nproc)"
```

The setup script defines two variables: `$ROOTCOREDIR` and `$ROOTCOREBIN` (the later since RootCore -00-01-00). The first points to the RootCore directory itself, the second to the directory at which the RootCore scripts put their data files and link the binaries they create. Typically they both point to the same directory (which is a safe and sensible default), but if you want you can make `$ROOTCOREBIN` point anywhere you like. Just define it as an environment variable before you setup RootCore:

```
export ROOTCOREBIN=/mypath
source RootCore/scripts/setup.sh
```

or if you are using a c-shell:

```
setenv ROOTCOREBIN /mypath
source RootCore/scripts/setup.csh
```

Please note that so far I have to see a good reason to do this, but if you want it, there it is.

If you want to you can also define the environment variable `$ROOTCOREOBJ` to point to a directory that will then contain the object files. This is mainly meant in case that your code is sitting on some network drive that either has limited space or is particularly slow. I personally use it for putting more code inside my AFS directory. Note that this may not work with certain other features like rel submission to PROOF or the grid. Also note that some object files may still end up in your package directories.

If you want to, you can also set the environment variable `$ROOTCORE_VERBOSE` to 1, which will make the output of RootCore compilation very verbose. This is mainly useful if you suspect that there is something going wrong in the way RootCore compiles your packages.

If you decide you no longer want RootCore setup there is a special script you can source. This will undo anything done in the setup script. It is currently only available for bash shells (if you need it for csh, send me an email). Note that this one you have to source:

```
source $ROOTCOREDIR/scripts/unsetup.sh
```

Setting up Root on Lxplus and Tier 3 Sites

The preferred way to set up root on lxplus and many Tier 3 sites is to set it up from CVMFS:

```
export ATLAS_LOCAL_ROOT_BASE=/cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase; source ${ATLAS_LOCAL_R
```

Setting up RootCore on MacOS

On MacOS it is necessary to disable System Integrity Protection. There are just too many things that won't work otherwise to make it even worthwhile to fixing this.

If you haven't done so already, install the command line tools (you may have to install xcode first):

```
xcode-select --install
```

Or if you upgraded your system, you may have to run (didn't try this one myself):

```
xcode-select --switch
```

Compilation Commands

To compile all your packages, just call:

```
rc compile
```

If you just want to compile an individual package, you can also call make for that package directly:

```
rc compile_pkg MyPackage
```

Or you can call make yourself:

```
cd MyPackage/cmt
make -f Makefile.RootCore
```

Sometimes you get a lot of error messages at once, garbling the output. To fix that, you can pass `--single-thread` into `compile` which then compiles one file at a time:

```
rc compile --single-thread
```

It should be noted that `compile` and `compile_pkg` do some additional steps and checks that the make for an individual package does not. So if you use make it is good practice that once you are finished with your changes to an individual package you should call `compile` or" compile_pkg" again to make sure that everything is still good and works with all the packages.

There are a couple of options for compilation:

- `--continue`: request compilation to continue as far as possible, even if an error occured.
- `--log-files` (`compile` only): create log files, one for each package. If `$ROOTCORELOG` is set, it will place them into the corresponding directory, otherwise in the current directory.

If you add another package or change a package makefile, you should have RootCore rescan for packages. You have to execute this from the root of the directory containing all your packages (called `packages` in the initial example), since it will just go through the current directory hierarchy:

```
cd packages
```

```
rc find_packages
```

Scanning for packages also takes care of handling package dependencies etc., so it is important that you rerun this when you change your package makefile. This implies that you can store your packages in a cmt like directory structure if you want, or you can just keep them all in the same directory, which is the RootCore default. You can add additional release parameters to find_packages if you want. In case you have packages that are nested inside each other (which is **not** recommended) you need to add the parameter `--allow-nested`. If you want find_packages to ignore missing packages in the dependency list, you can add the extra parameter `--allow-miss`.

Normally RootCore keeps track of all your file changes and only recompiles what is needed. However, sometimes you will want to remove all the old object files from your installation to force their rebuild, e.g. when you changed root version or compiler flags. To do so, just call:

```
rc clean
```

You can also do so for an individual package:

```
rc clean_pkg MyPackage
```

Sometimes all you want to do is to remove the dependency files (`.d` files). This mostly happens when you remove individual header files:

```
rc clean_dep
```

In general RootCore will try to find all the appropriate compiler flags for you (including `-g` for debugging symbols). However, in some cases you may want to add some extra ones. To do so, simply specify the extra flags in the environment variable `CXXFLAGS`:

```
export CXXFLAGS=...
```

Please note that to recompile all your packages with the new flags you have to call `rc clean`. Also note, that if you can't compile without specifying additional flags it should be considered a bug. This is to fine-tune the compilation not to fix it. Finally, though I don't have an example what to use it for, you can also specify linker flags via `LDFLAGS`.

Package Management

RootCore contains a very basic package management on top of SVN. To use it, you put a list of packages into a file (called packages.txt on this page). It is nothing more than a text file of all the SVN repositories for all your packages. Empty lines, and lines starting with a `#` are ignored. To simplify writing this file, the following abbreviations are allowed:

- If a line starts with `atlas`, I automatically pre-prend `svn+ssh://svn.cern.ch/reps/`. This allows to shorten lines considerably.
- If a line ends with `/tags`, I automatically determine the latest tagged version and use that. This allows to keep up with changes in the package, while at the same time avoiding the instability of using the trunk version.
- If you have the environment variable `$CERN_USER` defined, this user name will be used when checking out packages.

The general idea is that for your analysis (or physics/performance) group you can have one such package list, containing the actual versions that are recommended for your analysis. This package list is ideally contained in a package of its own, so that it can be easily updated and distributed as needed. It is recommended that such a package should not contain any actual code, but it may contain other pertinent data, like the list of datasets

to be used. This is the RootCore analogue of an ATLAS release.

There is a list of all packages that I am aware of distributed with RootCore itself (`RootCore/all_packages`), but that is not really recommended for everyday use. For one thing it contains more packages than you need, increasing build times. For another, it only contains trunk or latest tagged versions, while you may want more stable versions for some packages. It is mainly there to allow me to check whether all packages still compile. If you want to get on that list, please send me an email.

There are a number of commands. First of all, when you want to checkout all the packages from your list, just call

```
rc checkout packages.txt
```

This will just check out all packages into the current directory. If a package is already there, it is ignored. There is also a `checkout_pkg` command that will check out an individual package. If you are using an analysis release and call `checkout_pkg` with just the package name, it will check out the package in the version used in the release.

If your package list was updated, you probably will want to update your packages to the version from the list:

```
rc update packages.txt
```

This will update all your packages, to the version from the list. If you currently have the head version of a package checked out and want to change to a tagged version, RootCore will give you instructions on how to do so manually. This is in order to allow you to check-in any changes before accidentally overriding the checked out version. You can also call `rc update` without a package list and it will only update update packages in the head version to the latest version.

If you want to know which version of the packages you have checked out, you can call

```
rc version
```

This will go through all configured packages and provide you a list of packages that can be used as a `packages.txt` file. When you request help by email you should include that list as well. Please note that this will not work unless you have called `rc find_packages` or `rc build`.

If you have done some development work and want to see what changed, you can call

```
rc status
```

This will run `svn status` in all your RootCore packages. This is particularly useful if your code is spread out over multiple package, since it will allow you to see immediately if you had to make changes to any packages besides your current package. This command also doesn't require any network connections, so it is very fast. There is however the option to compare to what you would get if you ran update, simply pass `-u` to the status command.

If you want to have a history of what changed from tag-to-tag for a given package, you can call

```
rc tag_log atlasoff/PhysicsAnalysis/D3PDTools/RootCore
```

where you can replace the package name with whatever package you are interested in. Please note that some packages have a ChangeLog file that contains that information, and some packages do not contain meaningful SVN comments on which this method relies.

Code Commit and Tag Creation

If you are ready to commit your code (and possibly make a new tag of your package), you can use

```
rc tag_package "message"
```

This will update your package from SVN to make sure you are at the head of the package. Should you see any changes there, you should abort, recompile and run your tests, then call tag_package again. If you are not in the head of your package it will ask for confirmation before switching.

Then it asks you whether to add the message to the ChangeLog file (creating it if necessary). You have the option to skip this if you already added the message (or don't want a message/ChangeLog file). Before you type yes here, check that this is actually the message you want. The message will be automatically formatted and padded with the date, the author and the tag it goes in to. The author string can be specified via the environment variable ROOTCORE_AUTHOR or by passing it with --author into the command.

Next, if there are any changes it asks whether you want to commit them. Take a quick look at the printout from the update to make sure you added all your files, and that there weren't any incoming changes from SVN. If that seems Ok, type yes; otherwise abort make the necessary adjustments and call tag_package again. Next it asks you whether you want to create a new tag, which you can skip if you are not ready for that. You can also wait at this point and check out the code on another machine and test it there before you type yes here.

If you changed multiple items, you can also specify multiple messages:

```
rc tag_package "message1" "message2"
```

This will create separate items in the ChangeLog file, while everything will be bunched together in one long message for the SVN logs.

By default tag_package assumes that you will just increase the tag-number of the highest tag by one, which is often a reasonable assumption. Sometimes you may want to jump version numbers however, or match your version number to some external version number. That can also be done:

```
rc tag_package --tag MyPackage -XX-YY-ZZ "message"
```

This will affect both the entry in the ChangeLog and the actual tag created. In some circumstances you may want to increase the tag-number of the second highest release by one, which can be done via the `--previous` command.

If manually answering all the questions seems tedious, you can also specify `--batch` and it will automatically say yes to all questions. Please note that this is intrinsically a very unsafe operation. It is mainly useful if you changed a base package and had to update a dozen dependent packages to track the changes (which otherwise is very tedious). In that case, make sure you call "rc status", "rc version" and "rc update" first, to see if everything is correct, and then you can process everything in a shell for-loop. Note that this still isn't very safe, especially if others are also checking in changes at the same time. You have been warned!

Miscellaneous Commands

If you have a PROOF farm, you may want to create a proof archive containing your RootCore packages. This is done through

```
rc make_par
```

This will create a single proof archive containing **all** of your RootCore packages and named `RootCore.par`. Since this isn't really the most convenient, so you can provide the name of the .par file as an argument. There is also the possibility to specify `--lite`, which provides a shell .par file that just references back to your local RootCore installation. As an alternative you can specify `--nobuild` which packs up libraries inside the .par file which avoids recompilation. However, for this to work you need to use the same architecture on your proof farm as you do locally. Within your code you then need to load the `.par` file:

```
proof->UploadPackage ("RootCore.par");
proof->EnablePackage ("RootCore.par");
```

Warning: Make sure that the name of your .par file does not match any file/directory in your current directory, e.g. if it is named `RootCore.par` you should not have the RootCore directory directly in your current directory. Otherwise your job will fail.

At any point you can also call the `build.sh` script used in the beginning. It can be used with or without an argument. If it is used with an argument it will use it as a package list for `rc checkout`. Then it will call `rc find_packages`, `rc compile` and `rc make_par`. If you didn't setup RootCore already it will also take care of configuring and setting up RootCore for you.

If you want to run unit tests supplied with the packages run

```
rc test_ut
```

which will run all unit tests and report which ones failed. If you pass the optional argument `--fast` it will instruct all unit tests that they should not run if they are particularly slow to run.

If you want to, you can simplify calling RootCore commands through the `rc` short cuts, e.g. you can compile your code by calling:

```
rc compile
```

If you want to, you can ask RootCore to remove all temporary build files:

```
rc strip
```

Or if you want you can also remove all debugging symbols from the libraries as well:

```
rc strip --remove-debug
```

In either mode, this is mainly useful for saving disk space, e.g. for batch submission.

Using RootCore

Once you have a working RootCore installation you will probably want to use it. Any executables build through RootCore will automatically be in your path, assuming that RootCore is setup. And from root you can load all the RootCore libraries with a single command:

```
root -l
[] .x $ROOTCOREDIR/scripts/load_packages.C
```

Please note that using this script is preferred to loading libraries manually, because it will take into account all the library dependencies, etc.

If you have a macro you can load it from inside the macro using

```
gROOT->ProcessLine (".x $ROOTCOREDIR/scripts/load_packages.C");
```

Note that this only works for interpreted macros, not for compiled macros, since this line will only be executed after it is compiled. Also note that if you use interpreted macros you need to have dictionaries for all the classes you want to use.

In `pyROOT`, the following should work:

```
from ROOT import gROOT
gROOT.ProcessLine (".x $ROOTCOREDIR/scripts/load_packages.C");
```

Enabling Cintex for Athena Releases

If you are using RootCore with an Athena release and you want to read D3PDs you may encounter problems due to the presence of the Reflex dictionaries in the Athena release. To alleviate that problem, you can try to setup Cintex. Within RootCore for use with `load_packages.C` the easiest way to do this is to checkout the package RootCoreCintex:

```
rc checkout_pkg atlasoff/PhysicsAnalysis/D3PDTools/RootCoreCintex/tags
```

This package will automatically enable Cintex whenever an athena release is detected, so it should be safe independent of whether you use an Athena release or standalone root. While it is unnecessary to use it in plain root, this can be useful if you want to use the same package list inside and outside of root.

Warning: It is **not** safe to link against RootCoreCintex in your own packages. That means you should not list it in PACKAGE_DEP, or as argument to `rc get_ldflags`. Also you should not use `rc get_all_ldflags` if you use this package.

Linking Binaries With Your Packages

If you want to use RootCore libraries within your packages you have two options. The preferred and recommended way is to convert your code into a RootCore package, thereby ridding yourself of the need to maintain your own build system. For that purpose follow the instructions below on how to create a package, and put the source file for your program as `util/*.cxx`. Note that you need to have a `LinkDef.h` file or other source file `Root/` or your compilation will fail.

Or if you already have a build system that you are happy with, you can just add the commands for linking to RootCore libraries. For that a script `rc get_ldflags` is provided. Suppose your package depends on `GoodRunsLists` and `MuonSelectorTools`, then you would add the following to the link options in your make file:

```
$(shell rc get_ldflags GoodRunsLists MuonSelectorTools)
```

Please note that this command is smart enough to realize that `MuonSelectorTools` depends on `PATCore` and automatically includes that library as well.

If you are not using make, but a shell script to control your build, add the following to your link statement:

```
`rc get_ldflags GoodRunsLists MuonSelectorTools`
```

If you want to link **all** your RootCore packages you can use the scripts `rc get_all_ldflags` instead (without arguments).

All the include directories are linked inside `$(ROOTCOREBIN)/include` so you need to add that to your include path. Some packages also define compile time options, that can be obtained through `rc get_cxxflags`. So add to the compile flags in your makefile:

```
-I$(ROOTCOREBIN)/include $(shell rc get_cxxflags GoodRunsLists MuonSelectorTools)
```

Or in the shell script version:

```
-I$ROOTCOREBIN/include `rc get_cxxflags GoodRunsLists MuonSelectorTools)`
```

Running on the Grid

warning: If you are using EventLoop, you shouldn't be using this. You should instead use the GridDriver or PrunDriver: https://twiki.cern.ch/twiki/bin/viewauth/AtlasProtected/EventLoop#Grid_Driver

Panda incorporates RootCore support since version 0.3.51. Just supply the `--useRootCore` option to `prun` and it should pack up all your RootCore packages together with the rest of the job, compile them in the build job and then set them up in the actual run jobs. If you use the same setup locally as is used on the grid you can additionally specify `--noCompile` (since panda 0.4.0), this will use a dummy build job that performs no compilation and just copies your job to the local storage element. There is also the older option `--noBuild` which completely skips the build stage and may be somewhat faster, but from an operational point of view `--noCompile` is preferred/mandatory. In particular if your RootCore installation gets too large `--noBuild` will just print an error message and refuse to submit.

For instructions on how to select the root version in prun check here.

Known issues:

- The directory you run prun from should **not** be inside a RootCore controlled directory. A fine choice is to make a submit directory at the same level as your RootCore and packages directories. Otherwise you will submit your RootCore packages multiple times. This will not necessarily cause problems with your job, but it will waste resources.
- Similarly if you send your own files/directories with the job, they should **not** include RootCore controlled packages, or you will again send duplicate files.
- don't try to use the RootCore setup, compile or build scripts on the grid. Panda already does that for you. At best your calls will be redundant, at worst it will break your job.
- there are often unexpected issues when running on the grid, so don't be afraid to ask for help.

Submitting Multiple Datasets with the Same Tarball

Submitting a job to the grid can be a fairly slow process. Particularly if you are submitting a RootCore job with a lot of packages. And if you have to submit many times for many different datasets it becomes even

more of a drag. A possible way to speed this up is to build a tarball of your RootCore installation once, and then use it over and over again as you submit the individual datasets.

First do

```
prun OPTIONS --outTarBall=tarball.tar --noSubmit
```

and then for each dataset

```
prun OPTIONS --inTarBall=tarball.tar
```

Testing Your RootCore Installation for Grid Submission

If you encounter trouble submitting to the grid, there is now a test script that will allow you to test the RootCore part of your grid submission locally. Start with a clean shell, setting up root and RootCore. Then run:

```
rc grid_test $ROOTCOREBIN /some-directory
```

The first argument here is the location of the RootCore package, the second points to a directory in which RootCore can create some temporary files for testing purposes. It is important that the second directory is an absolute path. As a third argument you may pass a script that you want to execute in the run job to test that your job is really working.

This will test that your current installation of RootCore can in fact create a copy of itself in another place. It will also test that it survives being moved around after each step, like what might happen on the grid. If these tests pass from the directory where you are submitting to the grid, you can be fairly confident that the RootCore submission should work. There are however a number of other issues in submitting, compiling and running your job that are not checked for. As optional arguments you can pass `--nobuild` to use no build job, `--nocompile` to use a build job without compilation, and `--noclean` so that it won't clean up the area (in case you need to inspect it). Another optional argument (that you are unlikely to need) is `--grid-release-test` which pretends all your packages are part of a release.

Working With Releases On The Grid

If you are using an analysis release on the grid, RootCore should automatically detect which packages are taken from the release and not submit them again, substantially speeding up grid submission. However, this does only work when the release is set up from cvmfs (as only cvmfs releases are guaranteed to be available at all sites).

If you want to try using the nightlies on the grid and are sure that the target site has afs support, you can also run with releases from afs, by setting the variable ROOTCORE_GRID_RELEASE_PREFIX accordingly:

```
export ROOTCORE_GRID_RELEASE_PREFIX=/afs
```

If you want to disable this feature complete (thereby copying the entire release into your submission tarball, you can do that as well:

```
export ROOTCORE_GRID_RELEASE_PREFIX=none
```

Using Grid Submission Scripts for Batch Submission

Many users these days have access to a local batch system. Some batch systems are set up so that the batch jobs can directly use the RootCore installation from the users home directory. In that case all you have to do is source your setup scripts from the worker nodes. Others are setup so that the user has to send source files or

binaries to the worker nodes to be installed on the local disk. If you are in the later situation you can use the scripts developed for grid submission for this purpose.

The more common case is that you have the same architecture on your submission node and on your worker nodes, so let's deal with that first. To copy your RootCore installation into a target directory use

```
rc grid_submit_nobuild submitDirectory
```

It is important that `submitDirectory` doesn't exist yet, or your job will fail. Once this copy is complete you will typically have to add some steering scripts and tar up the whole lot. The result is a tarball you can submit to the batch system.

Then on the worker node, you may have to unpack the tarball or it may have already been done for you. Once that is done, let us assume that the untarred directory exists as `submitDirectory` and that root is already set up. Then you can setup your RootCore installation using:

```
source submitDirectory/RootCore/scripts/grid_run_nobuild.sh submitDirectory
```

This will then automatically setup your RootCore installation on the worker node.

If you have to submit source files to your batch system, you work similar as above, expect for the following changes: You have to call `grid_submit.sh` and `grid_run.sh` instead of their =nobuild counterparts. And you have to run a build job, in which you compile your RootCore packages:

```
source submitDirectory/RootCore/scripts/grid_compile.sh submitDirectory
```

The results of that compilation will typically have to be packed up again to be submitted to the worker nodes.

Please note that in this case I really do want you to source the RootCore scripts (except for submit). This will set up RootCore for anything that you will want to do subsequently in the build or run job.

Centrally Provided Data Files

warning: This feature is very new, so any number of implementation details can still change. It also means that feedback would be highly appreciated.

Some tools that you might want to use may come with extra data files, e.g. calibration data, pdf data, etc. If those files are small they may be stored within the package itself, but in many cases they are too large for that. This means one needs an alternate way of accessing them. There should typically be a copy on cvmfs/afs that can be used, but not all machines will have that available. If there is no local copy available it may be feasible to download it from a central webserver instead. All of this takes some effort to set up, namely one needs to place the files in strategic locations and make a list of those locations.

To use this tool, one needs to checkout `RootCoreUtils`:

```
atlasoff/PhysicsAnalysis/D3PDTools/RootCoreUtils/tags
```

And then add it to the PACKAGE_DEP line in the RootCore of your package:

```
PACKAGE_DEP = RootCoreUtils ...
```

This implementation assumes that when you want to access a file, you provide a list of possible locations as well as a fallback URL. It will then go through that list sequentially and pick the first option that works. The list of files is provided in a single string separated by `":::"`, e.g. the following will first check for the PDF set inside athena and if it is not found there it will download it from the given URL:

```
#include [...] std::string path = RCU::locate ("'$LHAPATH/cteq66.LHgrid::http://www.hepforge.org/
```

Please note that this will only download the file the first time around. On subsequent invocations it would still use the previously downloaded copy. It will also avoid submitting the downloaded files to the grid, since the grid sites should already have a copy of the relevant files on cvmfs/afs. Though your mileage may vary.

In general this should be fairly foolproof. The main caveat, is that if you want to run on a batch system you should first run the command on an interactive node first. Not only is the behavior undefined if you try to download with multiple jobs simultaneously, having a hundred nodes try to download the same set of large files simultaneously can overload a webserver. To that end there is a shell interface that will do the same as C++ interface:

```
rcu_locate '$LHAPATH/cteq66.LHgrid::http://www.hepforge.org/archive/lhapdf/pdfsets/current/cteq66
```

Be warned that the shell interface is not quite as foolproof as the C++ interface. On success it should print out the location of the file, but it may be preceded with extra lines of diagnostic output. Use with care.

Adapting Your Package to RootCore

If you want to adapt your package to RootCore there are a couple of things that have to happen:

- your package needs to follow the directory layout for RootCore packages
- your package needs a `Makefile.RootCore`
- your package (probably) needs a `LinkDef.h`

A good way to start your RootCore package is to use `rc make_skeleton` to create the directories and files you need:

```
rc make_skeleton MyPackage
```

Package Layout

RootCore has pretty fixed ideas on where things have to go. It will also link certain directories inside the RootCore directory (or `$ROOTCOREBIN`), indicated by "(-> target)" in the following list:

- `cmt/Makefile.RootCore`: This contains your global make flags and is described in its own section below.
- `cmt/precompile.RootCore`: This contains your auto-configuration script and is described in its own section below.
- `cmt/link.RootCore`: This contains a script for linking files from external packages into the RootCore area. There is no independent documentation, but check out packages like RootCoreCLHEP or RootCoreRooUnfold for an example.
- `MyPackage/*.h` (-> `include`): Those are the header files. Please note that these really have to go into a directory with the same name as the package. Also note that changing the location of header files has repercussions for other packages that use your package.
- `Root/*.cxx`: Those are your source files that get linked into the library. If this is a dual-use package you should only put the source files here that do not rely on athena. Anything that relies on the athena EDM or core classes still should go into `src`.
- `Root/LinkDef.h`: This contains your CINT dictionary definition and is described in its own section below.
- `util/*.cxx`: Source files for executables that your package creates. Please note that there is a one-to-one correspondence between source files and executables and you can't link two source files into the same executable. Also note that you should choose a name that is unique in all of Atlas, or at least unique among the packages you are using.
- `scripts` (-> `user_scripts`): Scripts the package provides.
- `data` or `share` (-> `data`): Data files that are distributed as part of the package.
- `test/*.cxx`: Source files for executables that test this package.
- `test/ut_*.cxx`: Source files for unit tests of this package.

While running it also creates a bunch of directories inside the package (as needed):

- `obj`: Contains all of the object files, as well as the CINT dictionaries.
- `StandAlone` (-> `lib`): Contains the library file.
- `bin` (-> `bin`): Contains the executables this package provides.
- `test-bin`: Contains the executables for the test of this package.

Please note that if you want to avoid this directories showing up in your `svn status` commands, you can tell SVN to ignore them. To do so call the following on your package directory:

```
svn propedit svn:ignore .
```

and then add all the files you don't want SVN to care about. You can also do that for subdirectories that you want SVN to ignore.

Package Makefile

The package makefile is essentially just a collection of configuration options that define how your package gets compiled. It gets read both by make and by ordinary shell scripts. So if you try to do anything fancy here, it will almost certainly break something. The name of the makefile was chosen as `Makefile.RootCore`, to avoid clashes with any pre-existing makefiles. The makefile itself includes `$(ROOTCOREDIR)/Makefile.common`, which contains the actual make rules.

The easiest way to create the makefile is to call the `rc make_skeleton` script above. It will create a default makefile with all the currently supported fields. Please note that whenever you change this makefile you should rerun `rc find_packages` and possible also run `rc clean`:

- `PACKAGE`: The name of the package. This has to match the directory name of your package. If it doesn't match RootCore will generate an error during `rc find_packages`.
- `PACKAGE_PRELOAD`: A list of all the external libraries this package depends on. Popular choices here include `Hist` and `Tree`, which will add the root libraries for histogram and n-tuple processing.
- `PACKAGE_DEP`: A list of all the RootCore packages your package depends on. Listing a package here achieves two purposes: It generates an error if the package is not present and it ensures the other package is compiled before this one.
- `PACKAGE_TRYDEP`: Same as `PACKAGE_DEP`, only that it doesn't generate an error if a package is missing. This is mainly useful in conjunction with package auto-configuration (see section below).
- `PACKAGE_CLEAN`: A list of files that should be deleted when object files are removed. This is normally not necessary.
- `PACKAGE_PEDANTIC`: Request to compile this package to be compiled with the -pedantic (and related options), supported since RootCore -00-01-47.
- `PACKAGE_NOOPT`: If this is set to 1, the compiler optimizations are turned off for this package. If this is set to dict, it will only disable optimizations for the dictionary. Neither of these is normally not necessary, but there are some cases where compiler optimizations cause an unacceptable delay during compilation.
- `PACKAGE_NOCC`: If this is set to 1, no library will be generated for this package. This is mainly useful if the package only consists of a collection of header files, or is a front-end for including a third-party library.
- `PACKAGE_CXXFLAGS`: Extra compilation flags for this package.
- `PACKAGE_OBJFLAGS`: Same as above, but these flags also get propagated to all packages that depend on this one.
- `PACKAGE_LDFLAGS` and `PACKAGE_BINFLAGS`: Same as above, but for linker flags.
- `PACKAGE_LIBFLAGS`: Same as `PACKAGE_BINFLAGS`, but these flags get applied **before** any other link flags. This allows to set the library path, etc.
- `PACKAGE_REFLEX`: If set to 1, this package will use Reflex dictionaries instead of CINT dictionaries.

Please note that it is my general recommendation to abstain from hardcoding compiler and linker flags into your makefile. While these flags may work for you on your computer, there is no guarantee that they will work for other people on their computers. In some cases you can use `PACKAGE_PRELOAD` instead, in other cases you may have no choice. Please also note that using `PACKAGE_PRELOAD` normally does not link your dependencies into your library, which in turn means you should use `load_packages.C` to load the libraries into root instead of loading them directly.

For dual-use packages you need to update the requirements file as well, in order for cmt to find the source files in the new location. Basically I have no clue on how to write a proper requirements file, but adding these lines should work (feedback is appreciated):

```
library [package name]Lib "../Root/*.cxx ../src/*.cxx"
apply_pattern named_installed_library library=[package name]Lib
```

Some packages explicitly call their current standalone makefile for compilation. This is highly discouraged and the above solution is preferred. Also, while we generate a lot of dictionaries preemptively, that policy is highly discouraged when using Athena.

Generating Dictionaries

RootCore employs CINT dictionaries to make your classes available for interactive use with the ROOT interpreter, as well as to Python users. Currently our tendency is to provide CINT dictionaries for all classes in your package, since you never know if your package may be accessed from Python or interactive root. Please note that this is contrary to the Athena policy of generating only the dictionaries for the classes that it knows it needs.

To generate dictionaries you have to create a file `Root/LinkDef.h` that contains the dictionary definitions. You have to create it really at that specific point or RootCore won't find it. An example of a (shortened) version from the SampleHandler package:

```
#include <SampleHandler/Meta.h>
#include <SampleHandler/MetaData.h>
#include <SampleHandler/ToolsDiscovery.h>

#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;
#pragma link C++ nestedclass;

#pragma link C++ namespace SH;
#pragma link C++ class SH::Meta+;
#pragma link C++ class SH::MetaData<double>+;
#pragma link C++ class SH::MetaData<std::string>+;

#pragma link C++ function SH::scanDir (SampleHandler&, const std::string&);

#endif
```

The things to note here:

- You need to add include statements for all the header files you plan on using. And you need to add them before the `#ifdef` statement or the dictionary dependency file won't be properly remade.
- You need to add a dictionary for each class and function you want to make available.
- If you are using templates you need to add a dictionary for each specialization you want to use.
- If you have a namespace you should add that as well, although I am unsure what that does.
- If you don't want to access a class from python and don't want to save it to a root file you don't need a dictionary.
- If you want to save a class to a root file, you need to derive it from TObject (or another class that derives from TObject) and add a ClassDef /ClassImp for that class.
- If you derive a class from TObject you have to add a ClassDef /ClassImp and you should add a dictionary.

Using Reflex Dictionaries

Since RootCore -00-01-55 RootCore has experimental support for Reflex dictionaries. It is normally not recommended to use Reflex dictionaries, since those are less portable than CINT dictionaries, but if you want to give it a try, you can turn them on by setting `PACKAGE_REFLEX` to 1 in your `Makefile.RootCore`:

```
PACKAGE_REFLEX = 1
```

The use of Reflex dictionaries is analogous to Athena, i.e. you need to create a common include file for all classes in the dictionary called `MyPackage/MyPackageDict.h` and an xml selection file called `MyPackage/selection.xml`. You will probably also have to add Cintex to your package preload to make it work.

%w% **warning:** While Reflex is a perfectly fine system for building dictionaries, it is often a poor choice when writing RootCore packages, as it relies on `gccxml` which is an unsupported external in RootCore. You are typically better off using a CINT dictionary instead.

Compiling with -pedantic

The general recommendation is to compile your packages using the -pedantic option, which enables additional compiler options and errors, in the same way Athena does. For official packages it is expected that this will be made a requirement at some point. For this to work you need RootCore -00-01-47 or later. Also don't forget to run clean, or it won't recompile.

To enable it for a particular package set the `PACKAGE_PEDANTIC` option to 1 in your `Makefile.RootCore`:

```
PACKAGE_PEDANTIC = 1
```

Can't We Make -pedantic The Default?

Ok, I occasionally get the question whether I can make this the default. That is typically from people who use Athena and are frustrated that sometimes they get a RootCore package that wasn't compiled with the pedantic option and then fails to compile inside Athena. Well, if I made that option the default it would annoy more people in a more profound way. Plus it would be a serious strain on RootCore support, since it would break RootCore backward compatibility. However, with this option we can slowly phase in support for -pedantic without creating the same kind of problems. Simply point the package developer to this page and tell him to update his package (and RootCore installation) accordingly.

And for the people who live mostly in the athena world and think I'm simply cuddling my users: Making such a change would be like introducing a change to the panda clients that will break most existing athena releases, as well as many user packages, without offering any real tangible benefits to the user. How many people do you think will switch to the new clients and how many will stick with the outdated ones?

If you think nobody will update their package with this option, here is the good news: You can volunteer **your** time to contact the authors of the official packages and work with them to switch the packages over to the -pedantic option. I've had very good success with getting people to update their package by offering to help them with it.

Optional Dependencies

An optional dependency means that your package will use another package when it is available, but still work fine without it. For that to work you have to list that package in the `PACKAGE_TRYDEP` field of RootCore. In the package itself you then need to do a little magic so that you only use certain code when the package is there, e.g. if you have an optional dependency on the D3PDReader package, you would do the following to protect the include statement:

```
#include <RootCore/Packages.h>

#ifdef ROOTCORE_PACKAGE_D3PDReader
```

```
#include <D3PDReader/Event.h>
#endif
```

Dual-Use Packages

For dual-use packages (i.e. for packages that work in RootCore and Athena) the main part is to make sure that they have both a `requirements` file and a `Makefile.RootCore`. Include files should all go into the same directory, source files that are used by both RootCore and athena should go into `Root/` and source files that only work inside athena should go into "src/". If there are sections of a header or source file that should look different between Athena and RootCore, you can try something like:

```
#ifdef ROOTCORE // do something in Root #else // do something in Athena #endif
```

Using External Libraries

RootCore has its own mechanism for external libraries: For each external you have to install a separate package that interfaces it with RootCore. Typically these packages will first check if a local version of the library is found and if so they will just use that. If no local version is found they will download a source version of the package and compile it for you. Compiling these externals can take quite a bit of time, depending on what it is.

If you are working on the grid or a place like lxplus you may find it more convenient to use lsetup to setup the external instead of having RootCore compile them for you. This can save you some time (particularly for grid jobs). A lot of externals are already included in the analysis releases, in those cases it is **strongly** advisable **not** to check out the externals packages locally, as it will cause all release packages depending on that external to be recompiled (which could be the entire release).

The exact behavior of each of these packages is configured through the file cmt/options. Most importantly you can select there where it should look for existing versions of the package.

Using Boost Libraries

The boost libraries are quite popular among C++ developers. Unfortunately they are not installed on all systems and when they are installed they are not always installed in the same way.

Asg_Boost takes care of detecting boost and making it available to RootCore. If boost is not found, it will try to install a stripped down version of it. If you are using an analysis release it should already have the latest version of Asg_Boost included so all you have to do is add Asg_Boost to its dependencies. However, if you don't use an analysis release you will need to add the package to your local RootCore installation:

```
rc checkout_pkg atlasoff/AsgExternal/Asg_Boost/tags
rc find_packages
rc compile
```

If it doesn't find your boost installation, try setting `BOOSTINCDIR` and `BOOSTLIBDIR` to the directories containing your boost include directory and your boost libraries. Please note that BOOSTINCDIR should point to the directory containing the directory containing the "boost" directory, i.e. "include" instead of "boost/include".

* Asg_Boost will install only a subset of the boost libraries for you. If your favorite lib is not in there, you will need to contact me, so that I can add it.

warning: If Asg_Boost decides to install a local version of boost and you have a global boost installation in /usr/include (like on lxplus) it will pick up any header or library files not present in the local version, but

present in the global boost installation from the global installation. This will almost certainly cause you all kinds of problems.

Using LHAPDF Library

The LHAPDF library provides a common interface to reading parton distribution functions. For full documentation see http://lhapdf.hepforge.org/. Usually you can just pick it up straight from the analysis release by adding `Asg_Lhapdf` to your `Makefile.RootCore` and then running `find_packages` again. RootCore uses version 6 of LHAPDF which is written in C++ (older versions were written in Fortran). The main reason for that is lower memory consumption as the older versions of LHAPDF used so much memory that jobs would exceed the available memory when running on the grid (seriously).

To access the LHAPDF data in your code first initialise the needed PDF sets **once before the start of the event loop**:

```
LHAPDF::PDF* p0 = LHAPDF::mkPDF("CT10nlo/0");
LHAPDF::PDF* p1 = LHAPDF::mkPDF("NNPDF23_nlo_as_0119/0");
```

This loads the nominal PDF sets (index 0). If you need to determine PDF uncertainties you can do call

```
LHAPDF::mkPDFs("CT10nlo");
```

This returns a vector of pointers to the new'd PDFs for the named set (i.e. all the eigenvectors needed for the systematics). You can then just loop over the vector and call the reweighting function passing it the kinematic information in each event.

In the event loop:

```
for (e in all_events)
    double w = LHAPDF::weightxxQ(e.id1, e.id2, e.x1, e.x2, e.Q, p0, p1);
    // etc.
```

Also make sure you include the proper header file in your code:

```
#include <LHAPDF/LHAPDF.h>
```

For more information on the functionalities provided by LHAPDF6 see: PdfRecommendations and http://lhapdf.hepforge.org/

If you are not using an analysis release you need to check out boost and LHAPDF locally:

```
rc checkout_pkg atlasoff/AsgExternal/Asg_Boost/tags
rc checkout_pkg atlasoff/AsgExternal/Asg_Lhapdf/tags
rc find_packages
rc compile
```

RooUnfold

For more information on RooUnfold please see the official documentation as well as the statistics forum page.

You can access RooUnfold through the wrapper package Asg_RooUnfold. This is included in the analysis release, so usually the only thing you have to do is add `Asg_RooUnfold` to the `PACKAGE_DEP` field in your `Makefile.RootCore` file.

If you are not using an analysis release, you will have to check out Asg_RooUnfold yourself:

```
rc checkout_pkg atlasoff/AsgExternal/Asg_RooUnfold/tags
```

I fudged a little bit when it came to the header files, in that I put them all into their own directory, instead of having them top-level. So your include statements should read something like:

```
#include
```

CLHEP

CLHEP implements a variety of functions that can be useful for writing HEP code. For more information on CLHEP see the official documentation. This package is **not** included in the analysis release, as it is preferred for users to rely on the EIGEN package instead.

If you want to use CLHEP from RootCore, check out the Asg_CLHEP package:

```
rc checkout_pkg atlasoff/AsgExternal/Asg_CLHEP/tags
```

If you don't have a local version of CLHEP installed that this package can pick up you may run into compilation problems if you don't have cmake installed on your system. You can either pick that up via lsetup or by checking out the cmake external:

```
rc checkout_pkg atlasoff/AsgExternal/Asg_CMake/tags
```

CMake

CMake is a build system that is needed for some externals (notably CLHEP). Depending on your system, you may not have CMake installed or the version you have installed may not be working (notably SLC6 with ATLAS software setup). To fix that you can checkout CMake as a RootCore external:

```
rc checkout_pkg atlasoff/AsgExternal/Asg_CMake/tags
```

Please note that this package will first check for a working CMake version and if one is found it will skip installing CMake, so it doesn't cost much to have this package around when you have a working CMake.

Bayesian Analysis Toolkit (BAT)

For more information on BAT see the official documentation. If you want to use BAT from RootCore, check out the RootCore package:

```
rc checkout_pkg atlasoff/AsgExternal/Asg_BAT/tags
```

This package will first check if there is a system wide installation of BAT, and if it isn't found go through a list of locations as specified in Asg_BAT/cmt/options. If you feel there is a location that needs to be added, either add it on your own or ask me to add it for you. You can also specify a location through the environment variables BATINCDIR and BATLIBDIR.

If BAT is not found, the package will try to download and install BAT for you. If the tarball is already found in the data directory the download is skipped.

FastJet

For more information on FastJet see the official documentation.

FastJet comes pre-installed with the analysis release, so all you have to do is add `Asg_FastJet` to your

`PACKAGE_DEP` in the `Makefile.RootCore` file. However, if you do **not** use an analysis release you will have to check out the externals package locally:

```
rc checkout_pkg atlasoff/AsgExternal/Asg_FastJet/tags
```

So far the package doesn't have any default locations it checks for FastJet, instead it just always downloads and compiles FastJet for you. However, you can add new locations yourself if you want to.

This includes fastjet contributions (fastjet-contrib). Here are some more notes on the contributions:

- You may need to add appropriate flags to your PACKAGE_LDFLAGS, like -lNsubjettiness -lEnergyCorrelator etc.

Hans Boehm Garbage Collector

warning: This is a very new feature, which has received very little testing. However that should not stop you from giving it a try.

warning: So far there is no garbage collector support for athena. This means that any tool you write that depends on the garbage collector will not work properly in athena, or by extension production.

If you have worked in languages other than C++ you may already be familiar with the concept of a garbage collector and may be missing that feature in C++. But if you are not familiar with it, let's just say that it is meant to make programming easier by removing the need to place a `delete` for every `new` you type. Instead the garbage collector will occasionally scan the memory and release any objects that are no longer used, thereby eliminating the most common type of memory leaks. If you are worried about performance, let's just say that sometimes it's worse sometimes it's better. For more information see the official documentation.

To use it make sure that you are using `RootCore-00-01-67` or later and then check out the package:

```
rc checkout_pkg atlas-krumnack/rcpkg/RootCoreGC/tags
```

This is a slightly modified version of the garbage collector, which will replace the default memory management for all your code. In order to make it work with root's mechanism for loading libraries you will have to start root a little differently, i.e. instead of

```
root file.C
```

you will have to call

```
rc root file.C
```

Also, note that this does not completely eliminate all memory leaks that you may encounter, e.g. if you have an `std::vector` and you just keep adding elements to the end of it, they will not get released unless the vector gets released. Or if you create a new histogram on every event, the old histograms will not get released as long as they are associated with a file. Still, in most cases the garbage collector should do a fine job at simplifying your life.

Detecting Other Packages

RootCore typically gives the user a great deal of freedom in what packages he wants to install and which one he can do without. Now if you write a package, your package will often depend on other packages. Typically there is very little leeway in what packages your package depends on, if you need a certain package for your package to work then the user has to install that package. However, sometimes you don't really need a certain

package to make your package work, but if it is there you would like to add some extra code to integrate them better, e.g. a tool that tries to select good electrons doesn't need the D3PDReader, but if it is there it would probably want to support passing D3PDReader electrons straight into the tool.

Since RootCore -00-01-76 there is support to do exactly that. To use it one would first add the D3PDReader to the package `Makefile.RootCore`:

```
PACKAGE_TRYDEP = D3PDReader
```

If the D3PDReader package is present it works just like the `PACKAGE_DEP` file, i.e. it will make sure that the D3PDReader package gets compiled before your package and that the D3PDReader library gets linked with your library. If the D3PDReader package is not present, it will just silently ignore it, unlike `PACKAGE_DEP` which produces an error in that case.

Then in your code you do something like this:

```
#include // ... #ifdef ROOTCORE_PACKAGE_D3PDReader // code for D3PDReader support #endif
```

Package Auto-Configuration

Auto-configuration refers to the ability of packages to scan their environment and configure themselves accordingly. RootCore provides facilities that allow that since RootCore -00-01-00. Please note that these facilities haven't yet seen a lot of use in practice, so you may encounter some problems when using them. Also, this documentation is not the best, so look at some packages using it (EventLoop, NKCommon). Particularly the EventLoop package contains a good example of how to detect whether you have a D3PDReader installed and if it has all the capabilities you need.

The basis of auto-configuration is the file `cmt/precompile.RootCore`. If it exists, RootCore assumes this is a script and executes it before compiling the package. Since it gets executed every single time and auto-configuration is fairly slow you probably want to generate at least one output file and check for the existence of the file before doing anything. If you follow this strategy you should add the file to the `PACKAGE_CLEAN` field in your `Makefile.RootCore`.

Since auto-configuration often wants to change the compile time behavior, there are two scripts for getting and setting fields in the makefile. `rc get_field` and `rc set_field`. If you want to give the user the option to affect auto-configuration decisions, the recommended way is to add a file `cmt/config.RootCore` that you can then read (and write) with the same tools.

The main thing you want to do is test-compile code using various compiler settings. To do so, there is a script called `rc test_cc` which allows to do test compilations. It reads its source file from standard input (i.e. pipe it in there), and takes one parameter. If you need to debug the script, set the environment variable `$ROOTCORETEST_DIR` and it will put all its temporary files there and leave them there. The possible values for the parameter are:

- `compile`: run the test input through the compiler only
- `linklib`: try to build a library from the test input
- `link`: try to build an executable from the test input
- `run`: try to build an executable from the test input and then try to execute it

I'll add more examples in the future on how to use this feature effectively.

Unit Tests

For details on unit tests please the UnitTest page, linked from the SoftwareQuality page.

Binary Releases

The way releases are implemented in RootCore is that you point your local RootCore installation to another RootCore installation and it will pick up the packages in that installation that are not present in yours. In general RootCore tries to be smart enough to figure out which packages from the release have to be recompiled because they depend on local checked out packages. If you use a releases from the nightlies you should not set up your own version of Root, since that is done through the root glue package which is part of the release.

If you want to use a given release the easiest way to use it, is to create a RootCore work area from the release: If you want to make a new ROOTCOREBIN directory for a new workarea, you can create it using

```
(source $RELAREA/RootCore/local_setup.sh && rc make_bin_area newArea/RootCoreBin)
```

Please note that right now I am unsure as to whether it is a good or bad idea to name that directory RootCore (as opposed to e.g. RootCoreBin). On the plus side it will seem familiar, on the downside, it will make it hard for the user to check out his own version of RootCore if he wants to.

If you want to setup RootCore from the workarea the preferred way is to use:

```
source myArea/RootCore/local_setup.sh
```

If you have a pre-existing RootCore are you need to point find_packages to the ROOTCOREBIN of the base installation:

```
rc find_packages --obj-release BaseInstallation /RootCore
```

Or if you are using a release in source-only mode:

```
rc find_packages --src-release BaseInstallation /RootCore
```

If you don't want to remember to pass that argument each time you call find_packages, you can instead call set_release first:

```
rc set_release --obj-release BaseInstallation /RootCore rc find_packages
```

The upside and downside is that find_packages will now always use that base release until you call set_release again with a different setting.

If the release contains a new version of RootCore you will be requested to setup RootCore again in a clean shell.

If your installation involves a lot of packages that need to be compiled, you can tell find_packages to focus on a few packages (and the packages they depend on), e.g.:

```
rc find_packages --restrict pkg1 --restrict pkg2
```

This is mainly meant to shorten compilation times for larger projects, and it will work even if you do not use the release feature. If you have specified a restrict flag as part of your release and want to override it, you can do it like this:

```
rc find_packages --drop-restrict
```

This is mainly meant to test whether all packages in the release will still compile after the changes in your work area. This is a way of checking whether your code changes break dependent packages without having to

go through the nightlies.

For actually compiling the releases there is a new option for find_packages that allows missing dependencies:

```
rc find_packages --allow-miss
```

This allows find_packages to proceed, even if a particular package depends on a non-existent package. The compilation of that package will then of course fail.

If you want to modify a package from the release you can simply check it out and then modify it:

```
rc checkout_pkg MyPackage rc find_packages rc compile
```

Please note that this will also recompile all packages that depend on MyPackage, even though you haven't checked them out.

Other New Features

Determining the package name at compile time

If you need to determine the name of the package at compile time, e.g. for printing better formatted log messages, you can access it via ROOTCORE_PACKAGE.

Improved Unit Tests

You can now ask test_ut to give you a list of all available tests:

```
rc test_ut --list rc test_ut --long-list
```

The second version will print out the package name in front of each test, so that an external steering script can properly associate tests with packages.

You can also run test_ut with a particular test to run:

```
rc test_ut [test name]
```

Or, if you want to imitate running the tests automatically, try:

```
rc test_ut --auto [test name]
```

Besides the option `--fast` test_ut will now also accept the option `--slow` for including particularly slow unit tests. There is now also an option --no-delete, which will make sure that the unit test directories do not get deleted, so that they can be inspected as desired.

Root Glue Package

As part of the release mechanics there is now a root glue package. It allows to set up root automatically, and the main use of this feature is that it allows to select the root version through the tag collector (once glue packages for different root versions are available). If you use this glue package (or a release with this glue package) you should **not** try to set up root yourself.

It works mostly like a regular RootCore package in that you can simply check it out and find_packages will pick it up. The main difference there is that after the first time you call find_packages you have to call

```
source $ROOTCOREDIR/scripts/setup_external.sh
```

to setup root. In subsequent sessions this happens automatically as part of setup.sh, but this can not happen the first time since RootCore doesn't yet know about this package.

Internal Structure of the Glue Package

This glue package looks mostly like the regular RootCore packages for external software, but there are some differences: For one thing it has `PACKAGE_AUTODEP` set in `Makefile.RootCore` to make sure it gets processed before all other packages. More importantly it has a file `cmt/setup_external.RootCore.sh` that takes care of setting up root. This script needs to be able to safely handle multiple calls in the same session (since that can happen if the user adds more packages with external setup).

Inside of `cmt/setup_external.RootCore.sh` it will first check whether it can set up root using AtlasLocalSetup, and if that fails it will instead perform a source installation of root (using the standard RootCore mechanism for that). It is excepted that the production version will try a couple of other things first, before resorting to a source installation (if it does a source installation at all).

The package also currently doesn't have a `precompile.RootCore` since all that functionality is handled in `cmt/setup_external.RootCore.sh`, though in future versions it may be worthwhile to have a basic check that `setup_external.sh` really has been used.

Downloading Calibration/Data Files

There is a simple mechanism inside RootCore to access centrally provided calibration and data files. The basic idea is that you have a command like (replacing ... as appropriate):

```
rc download_file MyPackage /calibration.root /cvmfs/.../calibration.root /afs/.../calibration.roo
```

This will then check the list of given locations and pick up the file from the first location it finds, and as a last resort it will download the file from a web page. In the code the file can then be accessed as:

```
std::string bindir = gSystem-&gt;Getenv ("ROOTCOREBIN"); TFile file ((bindir + "MyPackage/calibra
```

To trigger the actual download from inside program, you can do:

```
if (gSystem-&gt;Exec ("rc download_file MyPackage /calibration.root /cvmfs/.../calibration.root /
```

In general it is advisable to use a checksum to ensure that the file download worked successfully. This is also very straightforward:

```
rc download_file --md5sum 62976a54d0ec279fbe8081fbf6a90d89 MyPackage /calibration.root /cvmfs/...
```

If you don't want to download the file to the local RootCore installation, you can instead download it to a different location indicated by `ROOTCORE_DOWNLOAD_DIR`. It should be safe to have multiple processes trying to download to the same location, i.e. I tried to remove all the race conditions from that.

You can also specify a search path for download areas via `ROOTCORE_DOWNLOAD_PATH` which can be used to avoid downloading the same file multiple times. I could automatically add certain directories to that path, if requested.

Visible Changes For Existing Users

This is an attempt to summarize what changes existing users may see when changing to the release aware versions of RootCore.

Most of the changes are related to setup:

- It is no longer necessary to do the initial setup via configure. It can still be called, but it does nothing.
- The check for the correct root version now happens inside "rc compile", not in setup.sh. To change root versions users now have to call "rc clean".
- When upgrading to the new version it is best/easiest to remove the old RootCore directory and check it out again from scratch.

During run-time there are a couple more changes:

- All object files are now stored inside $ROOTCOREBIN, instead of the package directories. To avoid confusion the old directories will be removed during compilation. This should be invisible to almost all users.
- All object files are now stored inside architecture specific subdirectories. This should be invisible to the majority of users, except the people who have hard-coded library locations inside custom makefiles. This practice is anyways discouraged, as users should use "rc get_ldflags" instead.
- Due to internal changes, compilation should now be substantially faster than before (I'd say about a factor 2-4).
- If the number of threads has not been specified RootCore will automatically use all available cores for compilation (as opposed to using just one thread in the past). Again, this should be a noticeable speedup.
- It is likely to be less robust than it was, as large fractions of RootCore have been rewritten. However, I am working actively on fixing any issues quickly as they get reported, and have been using the new version without noticeable problems for a while now.

When used in conjunction with releases, there are a couple more changes:

- Setup should then happen through rcsetup/asetup, which will set up the correct architecture, as well as setting the correct release.
- When running find_packages, it will automatically pick up the packages from the release.
- Packages from the release will be recompiled as necessary to accommodate locally checked out packages.
- Currently the case of using an architecture locally that is not defined in the release is not supported.

Known And Anticipated Issues

- the package JetSelectorTools currently does not support D3PDReader classes generated for NTUP_COMMON. The best way of handling this is to add a RootCore auto-configuration script that allows to detect what kind of D3PDReader is present and configure the package accordingly.
- I think there is still a stray auto-configuration script in PATCore, which is no longer needed and does (sort of) bad things when it is run.
- I haven't tested the .par files created with the make_par command yet, and I would anticipate that they won't work.
- there is still some issue to be sorted out on how to handle ROOTCORECONFIG inside local_setup.sh
- makefile_arch is stored inside rootcore_config not rootcore_config_$ROOTCOREARCH
- glue packages currently download the tarball into their source directory, not into $ROOTCOREBIN. actually, the whole download mechanism should first check whether the file is available from the base release before even attempting to download it.

- Until we have done the basic shakedown, don't expect that releases will work between different RootCore versions. They may, they may not. This restriction will go away in the future, but I don't want to carry along support for features only present in early development versions.
- there should be a extensive unit tests of RootCore itself
- It is not yet supported to check out a release by release name. Mainly because we have no convention of how releases are named and where they are located.
- Releases that take the form of a tag list in svn are not yet directly supported. The user would have to manually check out that tag list and then use the conventional mechanism for checking out the packages. Ideally RootCore would check out the tag list for the user in the background.
- There is quite a bit of cleanup work that needs to be done, or at least should be done behind the scenes.
- There is no check that the version of root in the base and in the current release is identical.
- There is no check that the user cleans out the relevant packages after changing releases. If you change releases you should call "rc clean" to make sure that your code gets recompiled. In the future that may be caught automatically.
- add an "rc doxygen" command for making doxygen documentation
- make_bin_area is way to chatty. fixed in next tag
- I should have separate bin areas for release files to avoid linking them on every compilation
- I should link include and data directories during find_packages to avoid linking them on every compilation
- there should be a check whether the user logged into a fresh shell after find_packages changed the RootCore installation used
- compile should print out the release information when a release is used. this is for debugging information.
- compile should check that find_packages has been called since the last set_release
- there is only one include directory for all different architectures. there may be some externals/packages for which this is not appropriate, i.e. they reconfigure their header files based on the architecture. Special mechanisms are needed to handle those.
- if typing "make -f RootCore " after running "rc clean" there is an indigestable error
- the download mechanism currently doesn't check in the download area of the release
- there seems to be no check for the proper root version when RootCore is first setup. this should still happen whenever possible.
- do something intelligible when using a binary release that doesn't support the current platform
- checkout_pkg should not require to have run find_packages first, of fail if there is broken SVN information
- there seems to be an issue that the externals don't properly recompile when their configuration changes
- clean and clean_pkg should clean if for all architectures
- find_packages should imply clean as well(?)
- checkout_pkg should add the package to the package list
- there should be a unit test to verify that make_bin_area makes a bin area that is usable without any extra commands
- it should be possible to tell find_packages to use RootCore from the release, so that it can be safely removed

Migration from RootCore to cmake

Migrating unit tests

In principle unit tests as developed in RootCore should also work in cmake as is, but you will have to register them in your `CMakeLists.txt` file. These are the only pages relevant for CMake at the moment I think:

- https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/CMTCMakeRosettaStone
- https://twiki.cern.ch/twiki/bin/view/AtlasComputing/SoftwareTutorialAdvancedCMake
- https://twiki.cern.ch/twiki/bin/view/AtlasComputing/SoftwareDevelopmentWorkBookCMakeInAtlas

It's probably best to consider them in this order.

RootCore only tests should be hidden like this:

```
if( XAOD_STANDALONE )
  # Only executed for AnalysisBase
endif()
if( XAOD_ANALYSIS )
  # Executed in both AnalysisBase and AthAnalysisBase
endif()
```

And all logical combinations of these are possible. Like:

```
if( NOT XAOD_STANDALONE )
  # Executed in all Athena based releases
endif()
```

See the CMake documentation for details on how to use the `if` function in all its glory if you need to do something more tricky than this.

For a unit test to be seen by ATN in a CMake nightly, one needs these two things:

- The test needs to be declared with `atlas_add_test(...)` in the `CMakeLists.txt` file. Like in:
  https://svnweb.cern.ch/trac/atlasoff/browser/Event/xAOD/xAODCore/tags/xAODCore-00-01-24/CMakeLists.t
- An XML file like this needs to be added to the test/ subdirectory of the package:
  https://svnweb.cern.ch/trac/atlasoff/browser/Event/xAOD/xAODCore/tags/xAODCore-00-01-24/test/xAODCo

The name of this file is not set in stone as far as I know, but the `[pkgName]_test.xml` convention should be a reasonable one. The format of these XMLs is described on: https://twiki.cern.ch/twiki/bin/view/AtlasComputing/ATNightTesting. But for AnalysisBase style unit tests we should really only use the formalism that's in `xAODCore` as well.

Known Issues

Common Problems

While I try to add good diagnostics to RootCore, so that it can tell you what goes wrong, there are a couple of issues that are traced back to some fundamental design choices and the way the compiler works that can not be automatically diagnosed:

- If you get a lot of undefined references to symbols that look like they are coming from root (or other third party) libraries, you are probably missing these libraries in your `PACKAGE_PRELOAD` field. For root libraries check out the class web page, e.g. for TTree look at http://root.cern.ch/root/html/TTree. There is a little floating box there that tells you the name of the library. In this case it says the library is libTree, so you have to add Tree to the `PACKAGE_PRELOAD` field. Please note that if you use another package that uses this library it may load it for you, which means that you may see the problem only in some environments and not in others.
- If you see a lot of undefined symbols for one of your own classes, chances are that you used the ClassDef /ClassImp macro, but did not define a dictionary for your class. Add that dictionary and the problem will go away. If that is not the case, check that you added an implementation for every function declared in your class.
- If you get a message that a header file from another package can not be found that package is most likely missing from the `PACKAGE_DEP` field. Add it there, rerun `rc find_packages` and everything should be fine.
- If you have the same package checked out twice (or two packages with the same name), `rc find_packages` will fail in a rather unusual manner.

Feature Requests

This is the list of features requested by users that have not yet been implemented:

- allow to split the packages over multiple PROOF archives. This would allow to load all the base packages once and only update the user packages for each run.
- allow `rc compile` to run on single packages instead of forcing to run on all packages.
- include the package with the package list into `rc update` and `rc checkout`. In practical terms, I should be able to checkout a RootCore "release" by giving the SVN location of the package and then having that package checked out first. If I update from a package list, I should have the option of updating the package list first.
- auto-detect number of cpus by scanning /proc. If somebody tells me how to do that on the mac, I'd love to do it there too.
- add a garbage collector
- support .dylib files directly in load_packages.C (MacOS only issue)
- rc find_packages: allow setting the location of the RootCore packages through ROOTCORESRC
- see if I can avoid relinking binaries if only the library changes
- make a quiet option that suppresses all non-error output
- protect grid_submit.sh from recursive inclusion of files when called from inside a package
- cause a package to rebuild if RootCore changed
- allow a specific directory (e.g. `submit`) inside each package that does not get submitted to the grid and from which grid submission can be performed.
- allow an option `--flat` to `rc find_packages` that speeds up the script if packages are all in one directory
- allow to checkout all dependencies for a package
- fix rc test_cc, so that it also uses all package supplied compiler flags
- investigate why profiling doesn't work on the mac
- check RootCore support for newer Mac compiler

- protect against RootCore containing +=
- allow compilation to copy all commands to screen if a certain environment variable (e.g. `ROOTCORE_VERBOSE`) is set. this is mainly meant for debugging.
- allow packages to request pedantic compilation
- in the grid submission scripts, protect against recursive copies by using the flag --no-inc-recursive
- trigger a rebuild when RootCore changed? I'm not sure about this. Often we change the make file for rather harmless reasons, that would then always trigger an automatic rebuild.
- add automatic retries to the svn scripts
- support automatically converting RootCore packages into dual-use packages
- allow running unit tests for a single package and as a make target
- add C++11 support
- add Garbage Collector: http://www.hpl.hp.com/personal/Hans_Boehm/gc/
- add a verbose mode that prints out all compilation commands
- add a quiet mode that prints out as little as possible
- remake dependency files that refer to non-existant files instead of producing an error (not sure how this is done)
- have rc check_dep check the PACKAGE_PRELOAD field as well
- have all uses of SVN do several tries to work around occasional failures
- update RootCore to support whatever build system cmt2 is using behind the scenes
- doublecheck that rc update works for packages that don't follow the standard Atlas SVN layout
- add support for LHAPDF6: http://lhapdf.hepforge.org/lhapdf6/ This mainly isn't done because I can't get it to compile on my machine.
- update should switch repositories if there are no local changes
- package management should have a --continue option to skip failures (beneficial for long package lists)
- it should be possible to include individual packages during grid submission. this can be useful particularly for cases when some packages depend on locally installed libraries.
- if the list of packages becomes too large then loading all libraries inside load_packages.C may become a memory hog. so far I'm not concerned, but when it happens let me know.
- add an option to RootCore that allows to add scripts to the bin directory (and thereby the path). it would probably involve specifying their names and patterns
- add a check in "rc compile" that prohibits overwritten existing links, thereby avoiding name clashes
- only set DYLD_LIBRARY_PATH on MacOS (determined via uname)
- the build.sh script isn't working properly

Major updates:
-- NilsKrumnack - 31-May-2011

Responsible: Main.unknown
Last reviewed by: **Never reviewed**

This topic: AtlasComputing > RootCore
Topic revision: r259 - 2017-07-06 - unknown