Table of Contents

LHCb Software Training: Basics

This hands on tutorial should be followed by all newcomers to the LHCb software environment. It covers the following topics:

Prerequisites

Before doing these exercises, you should look at the slides of the "Overview of LHCb applications and software environment" [.ppt] attached to this topic. Please feel free to update these slides if you modify them for a future tutorial session.

The instructions for these exercises assume that they are being executed on the CERN Linux interactive cluster lxplus. However, except for some details of the interactive environment, the exercises should work on any machine where the LHCb software is installed.

To get an account on lxplus, please contact the LHCb secretariat. If you already have an lxplus account, but were previously working in another experiment, you should make sure that your account is registered as an LHCb account, in afs group **z5** (see Exercise 1).

These instructions have last been checked against the DaVinci v33r1 (and Gaudi v23r5) environment. Please use this version of DaVinci or a more recent version.

Logging in to lxplus5 (lxplus slc5 cluster)

Remote access to lxplus is via the ssh protocol. Open a connection using the command:

```
ssh -X <username>@lxplus5.cern.ch
```

The `-X` option is needed to enable X11 connections when logging in to lxplus as a remote machine, to be able to open graphics windows and windows for the emacs editor. Alternatively, omit the `-X` option and include the following line in the ssh configuration file, `${HOME}/.ssh/config`

:

```
ForwardX11 yes
```

Some sites require *trusted X11 forwarding*, in this case it is not enough to have `-X` option, the option `-Y` is needed or the following line in `${HOME}/.ssh/config`:

```
ForwardX11Trusted yes
```

Choice of shell

By default, all new accounts for on lxplus are configured to use bash shell. An alternative shell is tcsh shell (which was the lxplus default until 15th March 2011), an enhanced version of csh. LHCb setup scripts are available for both the Bourne family of shells: sh, bash, ksh, zsh and (t)csh. There are subtle and not-so-subtle differences in the functionality of the different shells, and for scripting there are major differences between (t)csh and Bourne-type shells. A useful short summary can be found in:

- A.Taddei, Shell Choice: a shell comparison CERN report CN/DCI/162 (1994)

A user may redefine the default shell on lxplus by going to https://account.cern.ch/account/Management/MyAccounts.aspx, navigate to "Services" then Linux and AFS services (you will need to login using your CERN username and password).

In what follows, `<script>.[c]sh` is used as shorthand for:

- csh,tcsh: `<script>.csh`
- sh,bash,ksh,zsh: `<script>.sh`

LHCb software environment

The LHCb software environment is set up by sourcing the script `LbLogin.[c]sh`. For local installations the script resides in the directory defined by the environment variable `$MYSITEROOT`. For an installation using AFS, such as lxplus, it resides in

- `/afs/cern.ch/lhcb/software/releases/LBSCRIPTS/prod/InstallArea/scripts/LbLogin.[c]sh`

`LbLogin` sets a number of environment variables relevant for working with LHCb software, including setting default `PATH`, `LD_LIBRARY_PATH` and `PYTHONPATH` variables. It also sources `CMT.[c]sh`, which performs the setup for the Configuration Management Tool (CMT), used in LHCb for building software and for environment setup.

For users logging on to lxplus with an account in the LHCb group (**z5**), the `LbLogin` script is sourced automatically at startup by one of the group login scripts, which are all equivalent:

- `/afs/cern.ch/group/z5/group_login.[c,z]sh`
- `/afs/cern.ch/group/z5/group_[ba,tc,z]rc`

Which script is executed depends on the type of shell, and on how the shell was created (e.g. ssh or spawned from an existing shell).

LHCb flavour of emacs editor

Customisations to emacs designed to help LHCb users have been written, and are described in the LHCb emacs user guide. Even for users who prefer a different editor (vim, nedit, Joe's own editor, pico, nano, or other), the LHCb flavour of emacs is useful for creating skeleton algorithms and similar.

IMPORTANT : If you are working over a slow network connection, it is sometimes useful to work in a 'terminal mode', where a new X window is not opened. This can be done by running

```
emacs -nw
```

In this mode, the menu bars are not useful, so it is good to know a few key-strokes :

1. Open a file with control-X control-F
2. Save a file with control-X control-S
3. Exit with control-X control-C

Exercise 1

This is a simple exercise just to check your environment setup.

- Login to lxplus5.cern.ch
- Check that X11 connections are enabled, for example by opening emacs:

```
emacs &
```

- Check the group associated with your account by giving the command:

```
echo $GROUP
```

For an account in the LHCb group the result should be **z5**. If it is not, you can request a change to your primary computing group by going to the CERN Resources Portal-->List Services-->LXPLUS and LINUX-->Computing Groups-->select z5 from the primary group dropdown menu.

- Check which shell you are using, for example using:

```
echo $0
```

or

```
echo $SHELL
```

- Check that the standard LHCb environment has been set. If it has, then you will have seen a message at login time about your CMT settings, and the result of the command:

```
env
```

should contain a number of variables with prefix `LHCB` (`LHCBHOME`, `LHCBRELEASES` and so on), and with prefix `CMT` (`CMTROOT`, `CMTCONFIG` and so on). If this isn't the case then you need to setup the environment yourself.

- Setup the LHCb flavour of emacs and (optionally) the EDT keyboard emulation by adding the following lines to the `~/.emacs` file:

```
(load (expand-file-name "$EMACSDIR/lhcb"))
(load (expand-file-name "$EMACSDIR/edt"))
```

or

```
cp $EMACSDIR/.emacs ~
```

LHCb specific emacs commands are documented in the LHCb emacs user guide.

Organisation of LHCb software

LHCb software is based on a framework called Gaudi. This framework provides functionality useful in a wide range of contexts, such as file access, histogramming, message printing, and run-time configuration. The run-time configuration is based on job options.

Software in LHCb is developed in the context of ***CMT packages*** and ***CMT projects***.

- A *package* is a set of files, organised according to some directory structure, which provides some well-defined, circumscribed functionality. It is the basic unit of software development, meaning that all the files in a given package are tagged with the same version number and are released together. Changing just one file in the package implies releasing a new version of the whole package. Most LHCb packages contain code that can be compiled into one or more libraries.
- Packages with related functionality are collected in groups (or are said to be placed under a *hat*). The purpose of package groups is simply to group together, in the directory tree, all packages that are related in some way (for example all event model packages under the `/Event` hat, or all Rich packages under the `/Rich` hat).
- A *project* is a set of packages that are grouped together according to some functionality, for example all Gaudi packages (`Gaudi` project), most public LHCb header files and base class libraries (`LHCb` project), all reconstruction packages (`Rec` project), all packages specific to the simulation application (`Gauss` project). All packages within a given project are released as a single unit: changing just one package in a project implies a releasing a new version of the whole project. Choosing a given version of a project automatically selects a single version of each of the packages in the project.

Projects and packages have an associated version number, of the form `vXrY`, with `X` and `Y` integers, and this is incremented when changes are made in package contents. The hierarchy of the software organisation is then:

`<PROJECT>/<PROJECT>_<version>/<package group>/<package>`. For example

```
ls $LHCBRELEASES/DAVINCI/DAVINCI_v33r1/Phys/DaVinci/
```

Sub-directories typically found in an LHCb package include:

- `src`: source code;
- `<package>` (Sub-directory with same name as package): header files;
- `options`: job options associated with the package;
- `cmt`: information for CMT processing;
- `doc`: package documentation.

Package development in LHCb is currently carried out using the Subversion code versioning system (SVN). This defines a code repository, provides dedicated commands for moving code in and out of the repository, and keeps a record of all changes made.

With the standard LHCb environment, a user can obtain a copy of any package from the SVN repository using the command:

```
getpack <package group>/<package> <version>
```

The appropriate repository is selected automatically by this tool. If the version is omitted a list of all available versions for the specified package is given. The latest version of a package, (not necessarily a working version!) is always available as the package **head**. Access to the SVN repository requires ssh authentication, which is automatic on lxplus if you have a valid AFS token.

New releases of projects are placed in the release area after testing. The standard LHCb environment includes the variable `$LHCBRELEASES` pointing to the AFS release area.

Instructions for making package modifications and new packages available for inclusion in a release can be found in the documentation for Guidelines for SVN actions in LHCb. Web access is available to the SVN repository and to the release area.

As soon as a package update is committed to SVN, it will be picked up by one of the nightly builds the next night. Nightly builds are used to test new software prior to it being officially released. The nightly build status can be viewed at http://cern.ch/lhcb-nightlies/cgi-bin/nightlies.py. Advanced users wishing to pick up the latest updates can run their software against one of the nightlies, see instructions in the LHCb nightlies documentation

LHCb applications

The LHCb applications are projects that use the Gaudi framework to perform specific tasks. There is a project specific to, and with the same name as, each application. The main applications are:

- Gauss: event generation, using Pythia and EvtGen, and detector simulation based on Geant4;
- Boole: simulation of detector response (digitisation)
- Moore: high level trigger
- Brunel: event reconstruction
- DaVinci: event selection and data analysis
- Panoramix: graphical display of detector and event information
- Bender: python-based physics analysis

Additional projects contain software that is shared by several applications. The main ones are:

- LHCb: base classes and public include files
- Online: framework extensions and components for running in the online environment

- Lbcom: components common to all applications
- Hlt: components for online (HLT) reconstruction
- Rec: components for online (HLT) and offline reconstruction
- Phys: components for online (HLT) and offline analysis
- Analysis: components for offline analysis
- Stripping: components for offline event selection (stripping)

Exercise 2

This exercise is designed to give you some familiarity with the LHCb software organisation. It's fairly open-ended, in that you could spend a very long time exploring all of the LHCb packages. Until you're trying to find something - which is when knowing the organisation is a big help - it's probably not useful to spend more than 5-10 minutes looking around.

- Move to the LHCb release area: either

```
cd ${LHCBRELEASES}
```

or use the web interface to the release area: http://cern.ch/LHCb-release-area/

- Note the different project directories, and enter the project directory for any one of the projects.
- Note the different project versions available, and move to the directory for the latest.
- Note the different package groups, and have a look in two or more of these at what packages are available.
- Choose any two packages, and for each in turn look at which sub-directories are defined and have a look at the files that are in each.

Configuration Management Tool

CMT is used to specify the compilation and execution environment for packages within a project, the dependencies between projects and between packages, and how each package is built (for example, how to produce executables or libraries from source code). To allow this, a CMT package must have a `cmt` sub-directory containing a `requirements` file, where the relevant instructions are given. All LHCb packages developed in the Gaudi framework are CMT packages.

For a package with name `myPackage`, CMT itself defines an environment variable `MYPACKAGEROOT` that contains the package location.

The full set of commands that can be used in a `requirements` file is described in the CMT manual. Some of the most useful commands are shown below.

```
# Lines beginning with the symbol '#' are treated as comments.
# Blank lines are ignored.

#============================================================
# The following commands describe a package.
# Package name.
package myPackage
# Package version, should correspond to SVN tag.
version v1r0
# Structure, i.e. directories to process.
# A package must always have:
#    a cmt directory (for the requirements file)
#    a doc directory (for the release.notes file)
# It may have:
#    a src directory for source files (.h, .cpp)
#    a options directory for job options (.opts, .py)
#    a python directory for python scripts and configurables
```

```
#    a directory with the same name as the package, containing include files that should be visib
branches cmt doc src options myPackage
#============================================================
# The following command specifies packages which the current package needs for compilation and li
# The keyword is followed by package name and version and, where applicable, the package group in
# The version can usually be v* (i.e. any version) because the exact version is usually determine
use DaVinciTools v* Phys
#============================================================
# The following commands tell CMT what to build and how
# Build application with the specified file
application myApp ../src/myApp.cpp
# Build a library with the specified files
library myLib ../src/*.cpp
# Define link options and environment variables for loading component library
apply_pattern component_library library=myLib
# Define link options and environment variables for loading linker library
apply_pattern linker_library library=myLib
# Make the public include files visible to the whole project (i.e. copy them to the InstallArea)
apply_pattern install_more_includes more=myPackage
#============================================================
# The following commands define symbols (variables and aliases).
# They consist of a keyword followed by a symbol name, and possibly by a value.
# Previously defined symbols, including environment variables, can be used in value assignments b
# Treat symbol as a Make macro definition.This can be used to define a symbol as a local variable
macro MYCMTDIR $(HOME)/cmtuser
# Treat symbol as an environment variable. The symbol will be visible from the shell after execut
set LCGDIR /afs/cern.ch/sw/lcg
# Treat symbol as a path variable. The symbol will be visible from the shell after executing Setu
# When appending or prepending an item to a path variable not initialised in the requirements fil
# Set path.
path PYTHONPATH $(HOME)/python
# Append item to path.
path_append PYTHONPATH $(HOME)/myPythonPackage
# Remove item from path.
path_remove LD_LIBRARY_PATH $(HOME)/lib
# Prepend item to path.
path_prepend LD_LIBRARY_PATH $(HOME)/lib
# Create alias.
# The alias will be visible from the shell after executing SetupProject
alias myApp $(MYPACKAGEROOT)/$(CMTCONFIG)/myApp.exe
#============================================================
```

Most CMT commands, but not all, operate on `requirements` files, and must be given from a package's `cmt` directory. If a `requirements` file includes lines indicating that other packages are used, then these packages will be searched for in locations specified by the project environment, and the `requirements` files of each used package is processed in turn.

The CMT manual should be consulted for the full set of CMT commands, but a useful subset is given below.

- Perform package configuration - that is, create setup files and Make files:

```
cmt config
```

Note that when a copy of a package is obtained using `getpack` the package is configured automatically.

- After configuration, Make commands can be used from the `cmt` directory:
  - ♦ Build the package binaries:

```
cmt make
```

The output of the build operation is placed in a package sub-directory. The name of this sub-directory is given by the value of the environment variable `CMTCONFIG`.

- ♦ Build several files in parallel if possible:

```
cmt make -j
```

- ♦ Delete the package binaries:

```
cmt make binclean
```

- ♦ Delete all generated files, including copies made to the project `InstallArea`

```
cmt make clean
```

- Show all packages used by the current package:

```
cmt show uses
```

- Execute `<shell command(s)>` in `cmt` directory of current package and all used packages within the same project:

```
cmt broadcast <shell command(s)>
```

Multiple commands need to be separated by semicolons and enclosed in inverted commas. This can be useful, for example, to force rebuilding of all used packages in the user's area:

```
cmt broadcast "cmt config ; cmt make binclean ; cmt make"
```

- Execute an executable within the environment of the current project

```
cmt run gaudirun.py
```

Have a look at the options of `gaudirun.py`

Try

```
cmt run gaudirun.py -h
```

for help, or

```
cmt run gaudirun.py -v
```

for a verbose printout of all options.

To release a new package in the LHCb framework it must be imported into SVN so that `getpack` can find it. Follow the instructions here.

Exercise 3

This exercise introduces basic CMT functionality and demonstrates the use of the LHCb flavour of emacs to create different kinds of files.

- Look at the value of the `CMTPROJECTPATH` environment variable

```
echo $CMTPROJECTPATH
```

The search path for CMT projects was set up when you logged in. It includes a `User_release_area`, the `LHCb_release_area`, the `Gaudi_release_area` and the `LCG_release_area` (for external software)

- Create a CMT project directory in which to work.

```
setenvDaVinci v33r1
```

This command creates a directory `~/cmtuser/DaVinci_v33r1`, containing a file `cmt/project.cmt`. Any directory tree containing this file is called a **CMT project**.

- Look at the `cmt/project.cmt` file that was created

```
pwd
cat cmt/project.cmt
```

The `setenv<Project>` command places you inside your new CMT project, and creates a file that tells CMT that any CMT command issued from inside this directory tree should use the environment defined by the DAVINCI_v33r1 project, which is found somewhere on the `CMTPROJECTPATH` (look for it!).

If you want to work with a different application, or with a different version of DaVinci, simply re-issue the `setenv<Project>` command, e.g. `setenvBrunel`. Information about CMT projects, including tips and tricks for working with them, can be found here

- Create a new package in your `DaVinci_v33r1` project:

```
cd ~/cmtuser/DaVinci_v33r1
mkdir -p MyGroup/MyPackage/cmt
cd MyGroup/MyPackage/cmt
```

- Create a cmt requirements file

```
emacs requirements &
```

Look at the generated file and add the line

```
use GaudiAlg v*
```

Save the file. The `use` statement that you added tells CMT that the files in this package will require to compile and link against files in the `GaudiAlg` package.

- Try some basic CMT commands:

```
cmt show uses
ls
ls ..
cmt config
ls
ls ..
```

- Try creating some different types of files with emacs. Each time, make a small modification to the file and save it

```
emacs ../doc/release.notes &
emacs ../src/MyAlg.h &
emacs ../src/MyAlg.cpp &
```

- Use CMT to compile the files you just created, and make a library out of them

```
cmt make
ls ..
ls ../x86_64-slc5-gcc43-opt
ls ../$CMTCONFIG
```

- Similarly, make the debug version of the library (i.e. containing debug symbols for use with a debugger)

```
setenv CMTCONFIG $CMTDEB
cmt make
```

```
ls ..
ls ../x86_64-slc5-gcc43-dbg
ls ../$CMTCONFIG
```

Running LHCb Applications

When executing an application, the following environment variables are very important:

- `PATH`: search path for scripts and executables, e.g. `gaudirun.py`;
- `LD_LIBRARY_PATH`: search path for shared libraries.
- `PYTHONPATH`: search path for configurables

The environment in which you build and run an application is fully determined by choosing which CMT project you wish to work with, and the value of the `CMTPROJECTPATH` environment variable.

The `SetupProject` script removes the need to set these variables explicitly. It uses CMT to fully define the run time enviroment for a given version of an LHCb application. See also the SetupProject user guide and FAQ.

Once the environment is set, the application is executed by simply typing the executable name, giving the name of a *job options file* as argument. The job options control what the application actually does when it runs. Controlling what happens when the application is run means knowing how to add new code and understanding the job options.

In the remaining exercises of this tutorial we will use predefined job options files, and concentrate on different use cases for running applications in the LHCb environment.

Exercise 4

In this exercise we just execute the released DaVinci application, with default job options

- Start a new login shell, and execute the following commands:

```
SetupProject DaVinci v33r1
which gaudirun.py
echo ${DAVINCIROOT}
gaudirun.py ${DAVINCIROOT}/options/DaVinci-Default.py
$APPCONFIGOPTS/DaVinci/DataType-2010.py
```

The `SetupProject` command sets up all the necessary environment to run the application. Notice that you can execute these commands from any directory. The job output goes to your current directory, so you should choose a sensible current directory. Once you have chosen a version and set up the enviroment, the behaviour of the application is determined by the job options file that you supply as the argument of the `gaudirun.py` command.

Notice also that you can keep any number of different job options files in your own area, each describing a different configuration of the application (e.g. different cuts), without having to rebuild the standard application. This is recommended if your analysis can be carried out simply by modifying job options (i.e. if you do not need to write your own C++ algorithms).

`gaudirun.py` is a generic Gaudi application written in python. It is used as main program for all LHCb applications. The specific behaviour of the application is defined by the options file(s) provided as argument(s). Options files can be written either in the old Gaudi native syntax (.opts) or using python syntax. The python syntax is recommended.

Exercise 5

In this exercise we address the case where you also need to modify the application executable, typically by adding your own component library package.

- Set up the build environment for `DaVinci v33r1` and get your version of the application

```
setenvDaVinci v33r1
getpack Phys/DaVinci v33r1
cd Phys/DaVinci/cmt
```

- At this point you could modify the `requirements` file to add your own package to the list of packages to be accessed by the application. You could for example add the `MyGroup/MyPackage` that you created in Exercise 3. Note that the syntax is as follows:

```
use <package name> <version> <package group>
```

- Install the application

```
cmt make
```

- Finally repeat the commands of Exercise 4 to run the application. As in Exercise 4 you can run the same executable from any directory, with any number of different options files

Sometimes, it may be necessary to work with several different versions of an application. You should be very careful when you do this: if you have setup the environment for a given version of DaVinci using `SetupProject`, and then you move to a different version, you can end up with the environment variables pointing to the wrong version. At best you will execute the wrong version of the application or of your private package, but more likely you will get inexplicable crashes that are difficult to debug. If you need to switch versions, it is **highly recommended** that you open a new terminal window and repeat the steps of this exercise for the second version.

Exercise 6

This exercise demonstrates an alternative method to opening a new window each time you wish to change the version of project you are working with. This method is recommended if you switch versions frequently, for example if you are comparing the results of a new release with a previous one.

- In the existing window type

```
env | grep DAVINCI
echo ${DAVINCIROOT}
```

- Start a new terminal window, and execute the following commands:

```
setenvDaVinci v33r1
cd Phys/DaVinci/cmt
cmt make
which gaudirun.py
echo ${DAVINCIROOT}
cmt run 'gaudirun.py ${DAVINCIROOT}/options/DaVinci-Default.py
$APPCONFIGOPTS/DaVinci/DataType-2010.py'
env | grep DAVINCI
```

Notice that we didn't execute the `SetupProject` script, and that consequently the environment does not "remember" any of the environment variables required to run the application. The `cmt run` command executes

the application inside its own shell which is configured according to the environment and `requirements` file of the directory from which you are issuing the command. In contrast to Exercises 4 and 5, here you **have to** execute the application from the directory that contains the `requirements` of the application.

The advantage of this method is that now you can switch to a different project, containing a different version of the application (or even a different application) and simply re-issue the `cmt run` command to execute the new application, without having to worry about opening a new window and setting up a new environment.

Exercise 7

Another use of `SetupProject` is to give access to software external to LHCb. In this exercise we see how to set up the environment for the **Root** application

- Start a new terminal window, and issue the following command:

```
which root
```

The Root application is not accessible in the default lxplus login environment

- Now issue the commands:

```
SetupProject Gaudi ROOT
echo $ROOTSYS
which root
```

You now have the environment to execute the Root application. The version that you will execute is the version compatible with the latest release of Gaudi

- Now issue the commands:

```
SetupProject Gaudi v23r2 ROOT
echo $ROOTSYS
which root
```

This time the version of Root that has been selected is the version compatible with Gaudi v23r2

- Finally, issue the commands:

```
SetupProject Gaudi v23r2 ROOT -v 5.32.00
echo $ROOTSYS
which root
```

This time we have over-ridden the version of Root compatible with Gaudi v23r2, and selected explicitly a different version. This can be useful if you need to over-ride the version of an external library to use a different version from the one with which the application was released.

Exercise 8

CMT projects are very powerful tool for software development. You can find out more by reading the instructions for working with CMT install areas.

-- MarcoCattaneo - 04-Dec-2012

This topic: LHCb > LHCbSoftwareTrainingBasics
Topic revision: r94 - 2014-08-25 - MichaelWilkinson