

MUON  $g - 2$

# OFFLINE COMPUTING AND SOFTWARE MANUAL

V201402 APRIL 16, 2014



# Contents

1	<i>Writing Source Code</i>	5
1.1	<i>Top level CMakeLists.txt file</i>	5
1.2	<i>Directory level CMakeLists.txt file</i>	7
1.3	<i>Libraries produced from building</i>	9
1.4	<i>Using External Code (Linking)</i>	10
	<i>Index</i>	17



# 1

## Writing Source Code

Your source code lives within a git project checked out to your development area's `srcs` directory. The project has a top level directory<sup>1</sup> that contains the "top level" `CMakeLists.txt` file along with various subdirectories. Code with a common purpose should live in a particular subdirectory.<sup>2</sup> You may mix headers (`.h`, `.hh`), implementation (`.cc`, `.cpp`), and configuration (`.fcl`) files all in the same subdirectory.

### 1.1 Top level `CMakeLists.txt` file

The top level `CMakeLists.txt` file lives in your top level project directory (e.g. `srcs/gm2ringsim/CMakeLists.txt`). It has the main directives that tells CMake how to build your project.

Below is a representative top level `CMakeLists.txt` file.<sup>3</sup> The `mrbs newProduct` command will create a skeleton file for you.

```
1 # Ensure we are using a modern version of CMake
2 CMAKE_MINIMUM_REQUIRED (VERSION 2.8)
3
4 # Project name - use all lowercase
5 PROJECT (gm2analyses)
6
7 # Define Module search path
8 set( CETBUILDTOOLS_VERSION $ENV{CETBUILDTOOLS_VERSION} )
9 if( NOT CETBUILDTOOLS_VERSION )
10     message( FATAL_ERROR
11         "ERROR: setup cetbuildtools to get the cmake modules" )
12 endif()
13 set( CMAKE_MODULE_PATH $ENV{CETBUILDTOOLS_DIR}/Modules
14     ${CMAKE_MODULE_PATH} )
15
16 # art contains cmake modules that we use
17 set( ART_VERSION $ENV{ART_VERSION} )
18 if( NOT ART_VERSION )
19     message( FATAL_ERROR
```

<sup>1</sup> For example, the `gm2ringsim` project would get checked out to `srcs/gm2ringsim`, which is the "top level" directory.

<sup>2</sup> Examine `gm2ringsim` for more examples.

<sup>3</sup> There are five main parts of the file (roughly in order in the file)...

- Defining the project
- Loading CMake macros and setting the CMake environment
- Setting compiler options
- Specifying external packages that will be used
- Specifying subdirectories that contain a `CMakeLists.txt` file and, perhaps, code to build

```

20     "ERROR: setup art to get the cmake modules" )
21 endif()
22 set( CMAKE_MODULE_PATH $ENV{ART_DIR}/Modules
23     ${CMAKE_MODULE_PATH} )
24
25 # Import the necessary macros
26 include(CetCMakeEnv)
27 include(BuildPlugins)
28 include(ArtMake)
29 include(FindUpsGeant4)
30
31 # Configure the cmake environment
32 cet_cmake_env()
33
34 # Set compiler flags
35 cet_set_compiler_flags( DIAGS VIGILANT WERROR
36     EXTRA_FLAGS -pedantic
37     EXTRA_CXX_FLAGS -std=c++11
38 )
39
40 cet_report_compiler_flags()
41
42 # Set include and library search paths (the version numbers
43 # are minimum - if actual version of product is below specified,
44 # will get error)
45
46 # Everyone should include these
47 find_ups_product(cetbuildtools v3_07_08)
48 find_ups_product(art v1_08_10 )
49 find_ups_product(fhiclcpp v2_17_12)
50 find_ups_product(messagefacility v1_10_26)
51
52 # This project uses code from gm2ringsim,
53 # gm2dataproducts, and gm2geom
54 find_ups_product(gm2ringsim v1_00_00)
55 find_ups_product(gm2dataproducts v1_00_00)
56 find_ups_product(gm2geom v1_00_00)
57
58 # This project uses code from Root
59 find_ups_root(v5_34_12)
60
61 # Make sure we have gcc
62 cet_check_gcc()
63
64 # Macros for art_make and simple plugins (must go after
65 # find_ups lines)
66 include(ArtDictionary)
67
68 # Specify subdirectories to build
69 add_subdirectory( ups ) # Every project needs a ups subdirectory
70 add_subdirectory( DisplayDataProducts )

```

```

71 add_subdirectory( calo )
72 add_subdirectory( fcl )
73 add_subdirectory( test )
74 add_subdirectory( util )

76 # Packaging facility - required for deployment
77 include(UseCPack)

```

### 1.1.1 When you need to add/change a line in top level CMakeLists.txt

There are two situations for which you will have to alter the top level CMakeLists.txt file:

*If you add, delete, or rename a subdirectory* If you add a subdirectory, you must write a corresponding `add_subdirectory(dirName)` directive.<sup>4</sup> If you delete a directory, you must remove its corresponding `add_subdirectory` line. If you rename a directory, you must edit its corresponding `add_subdirectory` line to reflect the change. If you do not follow these steps, then some code may not build (without an error, so this mistake will be hard to find) or you may get an error when CMake tries to build a directory that no longer exists.

<sup>4</sup> The `add_subdirectory` directive tells CMake to go into that subdirectory and build code there. If you don't have the `add_subdirectory` then CMake won't look in the subdirectory at all.

*You use code from an external project* If you use code from an external project, you may need to add a corresponding `find_package` or similar line.<sup>5</sup>

<sup>5</sup> See section 1.4 for instructions.

## 1.2 Directory level CMakeLists.txt file

If your subdirectory (e.g. `srcs/gm2analyses/strawTracker`) has anything to build, has header files, or has further subdirectories, then it must have a CMakeLists.txt file (and a corresponding `add_subdirectory` line in the CMakeLists.txt from the directory above - see Sec. 1.1.1).<sup>6</sup> If your subdirectory has code to build, then the directory CMakeLists.txt file needs to have

```

1 art_make( )

```

A directory with no `.cc` or `.cpp` files has no code to build and so does not get an `art_make` line in the directory CMakeLists.txt file.

See the next section (Sec. 1.2.1) for arguments to the `art_make`. You should call `art_make` only once per CMakeLists.txt file.

If your subdirectory has header files, then those have to be copied to the release area when one runs `mrbs install`. To do that, you need a line in the directory CMakeLists.txt file with

```

1 install_headers( ) # No arguments

```

<sup>6</sup> The directory level CMakeLists.txt file is different from the top level CMakeLists.txt file. The latter is in your project top level directory, like `srcs/gm2analyses`. The former is in a subdirectory of that top level and is described in this section.

If your subdirectory has fcl files, then those need to be copied to the build area as well as the release area. There is some scripting involved to do that (put the following in the directory CMakeLists.txt file),

```

1 # install all *.fcl files in this directory to the release area
2 file(GLOB fcl_files *.fcl)
3 install( FILES ${fcl_files}
4           DESTINATION ${product}/${version}/fcl )
5
6 # Also install to the build area
7 foreach(aFile ${fcl_files})
8   get_filename_component( basename ${aFile} NAME )
9   configure_file(
10      ${aFile} ${CMAKE_BINARY_DIR}/${product}/fcl/${basename}
11      COPYONLY )
12 endforeach(aFile)

```

If your subdirectory has further subdirectories with code to build, then you need an `add_subdirectory( dirName )` line for each subdirectory.

### 1.2.1 Arguments to `art_make`

You can find documentation for `art_make` in its source code at

`$ART_DIR/Modules/ArtMake.cmake`. Basically, you need to specify what libraries to link against when you use external code.<sup>7</sup> If you don't use any external code, then you will have no arguments to `art_make`. It will tell CMake to build all regular source, modules, services, and input sources in the directory. If you do use external code, then you have four choices,

- If the source file using external code is a regular source (not a module, not a service, not an import source), then you need

```

1   art_make(
2       LIB_LIBRARIES
3       library1
4       library2 # if needed
5   )

```

- If the source file using the external code is a module source (e.g. `analyze_my_hits_module.cpp`) then you need

```

1   art_make(
2       MODULE_LIBRARIES
3       library1
4       library2 # if needed
5   )

```

<sup>7</sup> See Sec. 1.4 for how to tell if you are using external code.

- If the source file using the external code is a service source (e.g. `analyze_my_hits_service.cpp`) then you need

```

1  art_make(
2      SERVICE_LIBRARIES
3      library1
4      library2 # if needed
5  )

```

- If the source file using the external code is source code for an input source (e.g. `midas_source.cpp`) then you need

```

1  art_make(
2      SOURCE_LIBRARIES
3      library1
4      library2 # if needed
5  )

```

If you have a mixture of sources in your directory, you can string the calls together. For example,<sup>8</sup>

```

1  art_make (
2      LIB_LIBRARIES
3      ${ROOT_GPAD}
4      MODULE_LIBRARIES
5      gm2analyses_util
6      gm2analyses_strawtracker_util
7  )

```

Note that it does not hurt for code to build against a library that it doesn't need. So if you have five modules and only one needs to link against a library, put that library in the `MODULE_LIBRARIES` section. The one that needs it will link against it and the four that don't won't care.

### 1.3 Libraries produced from building

Every directory in your project that has code to build generates at least one library.<sup>9</sup> Say, for example, you have a directory called `gm2analyses/cal0`. Regular sources (not modules, services, nor input sources) get compiled and the objects go into a library called `libgm2analyses_cal0.so` (the name is the directory path with slashes replaced by underscores). Each module in the directory gets its own library. For example, if there is a module in that directory called `Analyze_Cal0_module.cc` then that code will go into a library called

<sup>8</sup> In the example to the left, regular sources get linked against Root's `libGpad.so` (see Sec. 1.4.2) and modules get linked against code built in the `srcs/gm2analyses/util` and `srcs/gm2analyses/strawtracker/util` directories (see Secs. 1.4.4 and 1.4.5).

<sup>9</sup> An important note, if your directory **only** has header files in it (should be a rare situation for code written by users), then no library will be produced (because there is no code to build - the header files are all included by other source code). You still need the directory level `CMakeLists.txt` file for the `install_headers()` directive, but do not do `art_make`. See Sec. 1.2.

`libgm2analyses_cal0_Analyze_Calo_module.so`. A similar thing happens for services and input sources. Therefore, one directory of code may produce several libraries. The `art_make` directive in the directory `CMakeLists.txt` file tells the build system to build code and make the corresponding libraries.

## 1.4 Using External Code (Linking)

Your code is almost never self-contained, especially when writing within the Art framework. You may use functions and classes from external libraries, like Root and Geant4. You may use algorithms, data products, and other functionalities from other projects, like `gm2ringsim`. You may use objects defined in other directories in your project. If you are writing an art module or service, you may use objects defined in the same directory, but in a different file from the module or service. All of these examples are “external code”.

Art uses *dynamic linking*, which means that the art executable (ours is called `gm2`) has very little code in it. Instead, it loads all of the libraries it needs at runtime. The other style is *static linking* where the executable has embedded in it all of the libraries it needs. Dynamic linking, as the name suggests, allows for flexibility with one executable able to load a variety of different libraries decided upon at runtime with the configuration file. There is, however, overhead in dynamic loading typically experienced as slow start-up time of the program. Static linking produces an executable with all of the libraries built in - so there is little flexibility in terms of functionality. But the start up time is much faster. Static linking typically leads to many copies of executables for the different functionalities, resulting in duplication of libraries that are in common. For maximum flexibility and non-duplication of libraries, art loads everything dynamically.

HOW DO YOU KNOW WHEN YOU ARE USING EXTERNAL CODE? An easy indicator is when you have a `#include` for a header file. For each `#include`, you need to think and perhaps add a corresponding link directive in a `CMakeLists.txt` file.<sup>10</sup> If you forget to link to a library that you need, you will get a missing symbol error when you try to run. This section will explain how to figure out these situations and actions you need to take.

<sup>10</sup> Remember the two types of `CMakeLists.txt` files: “top level” and “directory level”. The former (see Sec. 1.1) is the potentially big file at the top level of your project. The latter (see Sec. 1.2) is the smaller file in the directory with your actual source code files.

### 1.4.1 Includes for system headers and base art headers

System headers, like `#include <string>` do not require any special directives for linking. You get them for free.

Headers in `art`, `fhiclcpp`, and `messagefacility` do not require

anything in your directory level `CMakeLists.txt` file. The corresponding libraries are automatically loaded by the `art` executable. Your top level `CMakeLists.txt` file must contain the following lines,<sup>11</sup>

```

1  ...
2  cet_report_compiler_flags()
3  ...
4  find_ups_product(art v1_08_10 )
5  find_ups_product(fhiclcpp v2_17_12)
6  find_ups_product(messagefacility v1_10_26)
7  ...

```

<sup>11</sup> These lines add header file directories to the compiler include search path (e.g. without them, you will get a compilation error that header files cannot be found).

#### 1.4.2 Includes for Root headers

Including a header from Root is a little unusual because you do not have to give a path in the include, e.g. `#include "TCanvas.h"` (not `#include "root/TCanvas.h"`). If you include a header from Root, you will also need to link to the corresponding Root library. First, in the top level `CMakeLists.txt` file, you must have,<sup>12</sup>

```

1  ...
2  cet_report_compiler_flags()
3  ...
4  find_ups_root(v5_34_12)
5  ...

```

<sup>12</sup> That `find_ups_root` line adds the Root headers to the compiler include search path and creates CMake variables corresponding to each Root library.

If you look at the code for the `find_ups_root` CMake macro at `$(CETBUILDTOOLS)/Modules/FindUpsRoot.cmake` you will see lines like,<sup>13</sup>

```

1  find_library(ROOT_GLEW NAMES GLEW PATHS ${ROOTSYS}/lib
2  NO_DEFAULT_PATH)
3  find_library(ROOT_GPAD NAMES Gpad PATHS ${ROOTSYS}/lib
4  NO_DEFAULT_PATH)
5  find_library(ROOT_GRAF NAMES Graf PATHS ${ROOTSYS}/lib
6  NO_DEFAULT_PATH)
7  find_library(ROOT_GRAF3D NAMES Graf3d PATHS ${ROOTSYS}/lib
8  NO_DEFAULT_PATH)

```

<sup>13</sup> These lines define the CMake variables that correspond to Root libraries. You use them in the directory level `CMakeLists.txt` file to tell CMake to link against that library.

To determine the Root library you need, look up the Root object in the Root documentation at <http://root.cern.ch/drupal/content/reference-guide> (select the appropriate version of Root - usually the PRO version). Find the class name from the list and click on it. On the new page, on the very right hand side in a little greyed out box it will say the library that corresponds to that Root object. For example, if you `#include "TCanvas.h"` you need to link against the `libGpad` library. The CMake variable name will in general be the name of the

library, all upper case, with the `lib` replaced by `ROOT_`. So `libGpad` → `ROOT_GPAD`.

In your directory level `CMakeLists.txt` file, you will have the `art_make` directive. Add the appropriate CMake variable corresponding to the Root library you need. See Sec. 1.2.1 for where to put such items in the arguments. For example,<sup>14</sup>

```

1     art_make (
2         LIB_LIBRARIES
3             ${ROOT_GPAD}
4         MODULE_LIBRARIES
5             ${ROOT_TREE}
6             ${ROOT_TVMA}
7     )

```

<sup>14</sup> In the example left, regular sources are linked against `libGpad.so` while modules are linked against `libTree.so` and `libTVMA.so`.

### 1.4.3 Includes for GEANT headers

To include a header file from Geant4, requires you to have Geant4/ in the header path, for example `#include "Geant4/G4Track.hh"`. If you include such headers in your code, then you will also need to link against the Geant4 libraries. First, in your top level `CMakeLists.txt` file, you must have,

```

1     ...
2     cet_report_compiler_flags()
3     ...
4     find_ups_geant4(v4_9_6_p02)
5     ...

```

That line adds the Geant4 headers to the compiler include search path and creates the CMake variables `${G4_LIB_LIST}` and `${XERCESLIB}`. For any Geant4 header, just add those CMake variables to the `art_make` directive in your directory `CMakeLists.txt` file. See Sec. 1.2.1 for where to put such items in the arguments. For example, `srcs/gm2ringsim/calor/CMakeLists.txt` has, in part,<sup>15</sup>

```

1     art_make(
2         LIB_LIBRARIES
3             gm2geom_calor
4             gm2geom_station
5             artg4_material
6             artg4_util
7             ${XERCESLIB}
8             ${G4_LIB_LIST}
9         SERVICE_LIBRARIES
10            gm2ringsim_calor
11    )

```

<sup>15</sup> If you are curious, you can see where `G4_LIB_LIST` is defined in `$(CETBUILDTOOLS_DIR)/Modules/FindUpsGeant4.cmake`. `XERCESLIB` goes with Geant.

#### 1.4.4 Includes for headers in the project

The `#include` directive should include the path to the header file, including the name of the project even if the header is in the same directory as the source, though you could just give the header file name. For example, if `CaloHitSD.hh` is in the `gm2ringsim/calor` directory, then `CaloHitSD.cc`, when it includes `CaloHitSD.hh`, can do either

```
1 #include "CaloHitSD.hh"
or
1 #include "gm2ringsim/calor/CaloHitSD.hh"
```

The latter is preferred as it is clearer, but if you change the name of the directory, you must change the include as well.

If you have a regular source file and it includes a header that is present in the same directory, then you do not need to do anything to the `CMakeLists.txt` files. If you have a module, service, or input source file and it includes a header that is present in the same directory, then you need to link against the library for that directory. You do not need to add anything to the top level `CMakeLists.txt` file. To the directory `CMakeLists.txt` file, you must add the library. See Sec. 1.2.1 for where to put such items in the arguments. For example, `srcs/gm2ringsim/calor/CMakeLists.txt` has, in part,<sup>16</sup>

```
1 art_make(
2     LIB_LIBRARIES
3         gm2geom_calor
4         gm2geom_station
5         gm2ringsim_station
6         artg4_material
7         artg4_util
8         ${XERCESCLIB}
9         ${G4_LIB_LIST}
10    SERVICE_LIBRARIES
11        gm2ringsim_calor
12    )
```

If any source file uses a header that present in a different directory in your project, then you must link against that library. In the example above, code in the `gm2ringsim/calor` directory includes code from `gm2ringsim/station`, and hence `gm2ringsim_station` is present in the arguments of `art_make`.

An important exception to these instructions is if the directory with the header file contains **only** header files. In that case, that directory produces no libraries and you do not have to change the directory `CMakeLists.txt` file.

<sup>16</sup> In the left example, services in that directory are linked against the library that gets created from the regular sources, namely `libgm2ringsim_calor.so`. You can predict the name of the library by taking the source directory (e.g. `gm2ringsim/calor`) and replacing the slashes by underscores.

### 1.4.5 Includes for headers in other projects

If you have a source file (regular, module, service, or input source) that uses code from another project, then you need to do some work. An example here is code in `gm2ringsim` uses code from the `gm2geom` and `artg4` projects. The `#include` needs the path to the header file including project name, directory name and header name. For example, `#include "artg4/util/util.hh"`.

In your top level `CMakeLists.txt` file, you need a `find_ups_product` line for the project specifying the project name and a minimum version number. See Sec. 1.1 for an example.

In your directory `CMakeLists.txt` file, you need to list the library corresponding to the code you are using. See Sec. 1.2.1 for where to put such items in the `art_make` arguments. For example, `srcs/gm2ringsim/calor/CMakeLists.txt` has, in part,

```

1  art_make(
2      LIB_LIBRARIES
3          gm2geom_calor
4          gm2geom_station
5          artg4_material
6          artg4_util
7          ${XERCESCLIB}
8          ${G4_LIB_LIST}
9      SERVICE_LIBRARIES
10         gm2ringsim_calor
11 )

```

When the regular sources are built, they will be linked against code in `gm2geom/calor`, `gm2geom/station`, `artg4/material`, and `artg4/util`.

An important exception to these instructions is if the directory with the header file contains **only** header files. In that case, that directory produces no libraries and you do not have to change the directory `CMakeLists.txt` file. You still need to have the top level `CMakeLists.txt` file correct as described above.





# *Index*

add\_subdirectory, [7](#)  
art\_make, [7](#)  
  arguments, [8](#)  
CMakeLists.txt  
  directory level, [7](#)

  top level, [5](#)  
external code, [10](#)  
find\_ups\_geant4, [12](#)  
find\_ups\_product, [11](#)

find\_ups\_root, [11](#)  
install\_headers, [7](#)  
linking, [10](#)