MUON $g-2$

OFFLINE COMPUTING AND SOFTWARE MANUAL [GM2 V5_00_00]

December 17, 2014 Git version: v5_00_00_04-0-gb992c58 GM2-doc-1825

Contents

1 What is this document? 7

1.1 What code goes with this document? 7

1.2 Obtaining newer versions of this documentation 8

1.3 Obtaining the source for this documentation, contributing to it, and building it 8

2 Releases of gm2 11

2.1 `gm2 v5_00_00 -q e6:prof` 11

2.2 `gm2 v201402 -q e4:prof` 12

3 Getting started with gm2artexamples 13

3.1 Logging in and selecting a release area 13

3.2 Starting a development area 14

3.3 Checkout code 16

3.4 Building code 17

3.5 Testing 18

3.6 Running 19

3.7 Logging in again 20

3.8 Summary 21

4 *Developer Workflow* 23

5 *Using a Mac for Development* 25

5.1 *Installing CVMFS* 26

5.2 *Installing Xcode* 26

5.3 *Running Xcode* 27

5.4 *Configure Xcode for your project* 27

5.5 *Things you can do with Xcode* 31

6 *Getting Started with the Simulation* 35

6.1 *Geant4* 35

7 *Running the simulation* 39

8 *Writing Source Code* 41

8.1 *Top level* `CMakeLists.txt` *file* 41

8.2 *Organizing Source Code* 43

8.3 *Writing Modules* 43

8.4 *Writing Services* 44

8.5 *Writing Input Source Modules* 44

8.6 *Directory level* `CMakeLists.txt` *file* 44

8.7 *Libraries produced from building* 46

8.8 *Using External Code (Linking)* 46

9 Things You May Do in Your Code 53

9.1 Dealing with parameters 53

9.2 Readling enviornment variables 53

9.3 Throwing an exception 53

9.4 Finding a file 54

10 Frequently Asked Questions 55

Index 59

1

What is this document?

This document is meant to be a user's manual to the Muon g-2 offline and simulation software and computing system.[1] This document is a PDF file, so it is trivial to search and you can copy it to your computer/tablet/phone/watch and read it anywhere including your office, in meetings, on the plane, in the tub, etc. It is also generated by a git repository using the same build system infrastructure as our code base, so it is easy to version itself and keep in sync with code versions. We support writing sections directly in LaTeX (which you probably already know) and in Markdown (like LaTeX, but simpler). Finally, there's a special script that can run shell commands and put the output directly in the document (no cutting and pasting).

[1] This document replaces the documentation we had in the Redmine Wiki because the Wiki was hard to edit and keep up-to-date, hard to sync with versions, hard to search, and required a network connection.

The idea is to have documentation that is easy to read, easy to write, and easy to keep up to date. All links in the document are click-able in your PDF reader.

One nice thing about having Wiki pages was that each page can be short and so the documentation looks manageable, until you try to find something. The problem with one big PDF file is that it will be big and will look overwhelming. Remember to read the section titles carefully and just read what you need. Furthermore, all of the links to sections (e.g. in the table of contents) are live and will allow you to navigate the file easily. Nearly every PDF reader has a back button to take you back to previously read pages (back-traversing links if necessary); it will probably come in handy.

1.1 What code goes with this document?

The title of this document states the corresponding version of `gm2`. `gm2` is the "umbrella" product that specifies a release. For example, this version of the document goes with `gm2 v5_00_00`. On the bottom of the title page is the git version information for this document itself. For this version, it reads `v5_00_00_04-0-gfac1db0`. There are three

or four parts to this description, separated by *dashes* (not underscores; the underscores are part of the version). The first part corresponds to the `gm2` version, with an additional two digits at the end since the documentation may be updated more often than the g-2 code. This version should be the git tag of this document. The second part is the number of commits past the tag. If it is non-zero, then there are untagged changes. The third part is `g` followed by the git hash of the commit corresponding to this document (e.g. `0be91c0`). All of this could be followed by `-dirty`, which means that this document comes from source files with uncommitted changes.[2]

[2] Official documentation has zero for the second part (number of untagged commits) and no `-dirty`.

1.2 Obtaining newer versions of this documentation

The latest official version of this documentation is in GM2 DocDB as GM2-DOC-1825.[3] If you want look at an old version of the documentation, you can do that in DocDB, or better, build the specific version you want, as described in the next section.

[3] DocDB uses its own versioning scheme (just a sequential number) which does not correspond to the `gm2` release.

1.3 Obtaining the source for this documentation, contributing to it, and building it

To get the source,[4] follow the instructions in section 3. When you get to section 3.3, instead of checking out `gm2artexamples`, checkout `gm2swdocs`. You will be in the `develop` branch. If you want to checkout a particular tag, branch, or hash, you can do that with the `git checkout` command. For example,

[4] *Note:* The program `pandoc` at http://johnmacfarlane.net/pandoc is used to convert markdown and other file formats to LaTeX. It is part of our g-2 release for SLF6 machines. See below for installing it on your own machine.

```
git tag                      # Show all of the tags
git checkout v5_00_00_02  # Check out sources for this tag
```

You can also do `git checkout` on a git commit hash value to checkout the sources for that particular commit.

1.3.1 Changing and adding to documentation

If you want to change or add documentation, you should start a feature branch with `git flow feature start <your_branch_name>`. You can then alter or add your own documentation. When you are ready to complete your feature branch, send mail to `gm2-sim@fnal.gov` and let people look at your changes first.

There are several directories in `gm2swdocs`. You should not need to alter anything in the `Modules` nor `ups` directories. The former contains cmake macros needed for building the source files into PDF. The latter is for the build and release system. The other directories, `latex`, `markdown`, `bashmd` is where you'll put your documentation or make changes.

The `latex` directory has files in LaTeX as well as some LaTeX infrastructure files. The most important file in there is `manual.tex`, which is the main driver file for this document.[5] All other parts come in with an `\include{filename.tex}` command, but this is handled automatically by a `cmake` variable (you won't see the `\include` lines in the file). If you add your own LaTeX file in the `latex` directory, follow instructions in `srcs/gm2swdocs/CMakeLists.txt`.

[5] We are using a document class based on "Tufte" documents, where notes and captions go into the wide right margin. Please see the existing LaTeX files for examples.

The `markdown` directory has files written in the Markdown format and converted by Pandoc. A Google search on Markdown will give you lots of information. The Pandoc variant of Markdown is described in http://johnmacfarlane.net/pandoc/demo/example9/pandocs-markdown.html. See existing files in this directory for examples. If you want to write something quickly and do not need fancy LaTeX, then Markdown is the way to go. If you add a file to this directory, you must follow the instructions in `markdown\CMakeLists.txt`.

The `bashmd` directory has files written in Markdown but also actually runs bash code with the output going into the document. The best file to look at for an example is `bashmd/gettingStarted_gm2artexamples.bashmd`. Again, if you add a file to this directory, see `bashmd/CMakeLists.txt` for instructions.

Pandoc understands many Wiki mark-up formats. If you have a favorite one, it is possible to add it to this document and have `pandoc` process it. Ask for help. If you are not passionate about mark-up formats, then please just use Markdown as it works very well.

1.3.2 Building the documentation

If you are on Mac, a Windows machine, or your own Linux machine, you must have installed a full TeX suite and `pandoc` on your system. See http://johnmacfarlane.net/pandoc/installing.html for installation instructions for `pandoc`. If you are on `gm2gpvm`, everything is installed there for you, but you must issue `setup pandoc`; see below.

Assuming your environment is set up (see above) then you need to do, once per session, `. mrb s` . If you are on `gm2gpvm`, do `setup pandoc` (it only works on SLF6, so use machines `gm2gpvm02-04`). Then you can do `mrb b` to build. Note that by default, files in `bashmd/` will *not* be built as they can take a long time. If you do want them built, then do `mrb b -DBUILD_BASHMD=1`. Also, `pdflatex` will run many times to ensure that references and table of contents are all resolved. If you make changes, only those changed files will be rebuilt on subsequent builds. If you see an error like `Cannot find PANDOC` and you are own `gm2gpvm`, then you forgot to issue the `setup pandoc` command.

The output PDF file will be in `$MRB_BUILDDIR/gm2swdocs/latex/manual.pdf`. On a Mac, you can view it with,

```
open $MRB_BUILDDIR/gm2swdocs/latex/manual.pdf
```

When you have completed your feature branch, send mail to `gm2-sim@fnal.gov` and await further instructions.

2
Releases of gm2

This sections describes the various releases of gm2.

2.1 gm2 v5_00_00 -q e6:prof

Note the new version numbering scheme. The plan is that new versions of art advance the first number. New versions of g-2 code advance the second number. Bug fixes for g-2 code advance the last number. This release is the fifth one for g-2 since time began, thus the `v5`.

`gm2 v5_00_00` has the following:

- `art v1_12_02` Release Notes
- `root v5_34_21b`
- `geant4 v4_9_6_p03e`
- `gcc v4_9_1` with `-std=c++1y` for C++14 features.

2.1.1 How to migrate from v201402 to v5_00_00

This section is not official yet - it will be when we merge the `g5` branches into `develop`

If you have a branch that works with `gm2 v201402`, then you will need to make some changes for it to work in `gm2 v5_00_00` as some parts of the build system have changed. If you are working on code we have in Redmine, and you can merge the `develop` branch onto your branch without breaking your code (you should be able to do that, but there may be special reasons why not), then do the following:

```
# Go to your source directory and check out your branch
# Check in all code you've been working on and push to Redmine

# Now merge develop onto your branch
$ git pull origin develop
```

If there are merge conflicts, then you will have to resolve them. Accept changes from `develop` for `CMakeLists.txt` files and `product_deps` as those will have the necessary changes.

If you cannot merge with `develop` or you are working on code not in Redmine, then follow the instructions here for migrating your code to `v5_00_00`.

You need to change the top level `CMakeLists.txt` file and the `ups/product_deps` file. For the top level `CMakeLists.txt` file, copy the current one to a backup file. You will need to copy things from it. You can download a new top level `CMakeLists.txt` file by doing,

```
$ curl -u user:pass -O https://cdcvs.fnal.gov/redmine/attachments/download/22294/CMakeLists.txt
```

replacing `user:pass` with either your Fermilab Services username and password or the generic g-2 usernanme and password. For example, if the username is "me" and the password is "you", then you would enter `curl -u me:you -O ... .`

Edit this new file and in two places copy in from the old file the indicated text. For example, you need to copy in the `find_ups_product` and similar lines and the `add_subdirectory` lines. Furthermore, towards the top of the `CMakeLists.txt` file you must enter the product name.[1]

[1] Remember we use the terms project, product, and package somewhat interchangeably

Now you need to similarly edit the `ups/product_deps` file. Like before, copy it to a backup file as you will need copy some things in from it. Download a new version by doing,

```
$ curl -u user:pass -O https://cdcvs.fnal.gov/redmine/attachments/download/22295/product_deps
```

Again, fill in `user:pass` as before. Edit that new file and copy the items in from the old file as indicated in the new `product_deps` (look for `<ADD ...>` in the file).

Now, you need to update versions in your dependencies. The way things are set up right now this is a bit tricky. Ask Adam for help.

2.2 gm2 v201402 -q e4:prof

`gm2 v201402` has the following:

More needs to go here.

3
Getting started with gm2artexamples

This section is a short tutorial to show you quickly how to get started by,

- Logging in and selecting a release (the latest)
- Starting a development area
- Checking out code (`gm2artexamples`)
- Building it
- Testing
- Running
- Logging in again

For this tutorial, we'll use the `gm2artexamples` product.[1] This is a good product to use if you are getting started.

[1] We use the terms *product*, *project*, and *package* somewhat interchangeably. All of our products live on the Redmine server, http://redmine.fnal.gov

3.1 Logging in and selecting a release area

Fermilab has several interactive virtual machines for use by the Muon $g-2$ collaboration. See here for more information about how to log in. Our releases (libraries, executables) are served by CVMFS.[2] CVMFS is already mounted on the Fermilab interactive VMs. If you have a Mac, you can install CVMFS yourself by looking here, and then use your Mac to develop code.

[2] CVMFS is a system that serves application code and updates automatically when new files are released.

Once you've logged into the machine, you need to select a release area. You *always*[3] need to do this step everytime you log in. If you are on a Fermilab interactive VM (`gm2gpvm01`, `gm2gpvm02`, `gm2gpvm03`, `gm2gpvm04`), you select the release area by doing,

[3] You need to do this step everytime you log in because you can use different release areas for the same development area, say, for example, if CVMFS is down or you are sharing a directory between your Mac and a Linux system.

```
$ source /grid/fermiapp/gm2/setup # On gm2gpvm machine
```

Note that `$` is the shell prompt (don't type it in).

If you are on a Mac or another system with CVMFS OASIS installed, you do,

```
$ source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup
```

Now we'll actually run it so you can see the output. This script will work on both Mac and gm2gpvm. You may want to put it in your .profile on gm2gpvm.

```
$ if [ -r /grid/fermiapp/gm2/setup ]; then  # Does /grid/fermiapp/gm2/setup exist?
$   source  /grid/fermiapp/gm2/setup   # We're on gm2gpvm
$ else
$   source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # We're on a Mac
$ fi
```

```
g-2 software

--> To list gm2 releases, type
ups list -aK+ gm2

--> To use the latest release, do
setup gm2 v5_00_00 -q e6:prof

For more information, see https://cdcvs.fnal.gov/redmine/projects/g-2/wiki/ReleaseInformation
```

You may want to put that source command in your ~/.profile file. Furthermore, if you are on a gm2gpvm machine, you should also put setup git at the bottom of your ~/.profile.

3.2 Starting a development area

Now that the release area is selected, you need to make a *development area*. The development area contains source code, build products, and a personal release area. You typically use a development area for a particular topic, such as adding a feature to the simulation or generating a plot for some study. You can have as many development areas as you want, but only one can be active at a time.

Make an empty directory and go there. If you are on a gm2gpvm machine, you should make an area in /gm2/app/users/<YOUR_NAME>.[4] You can put code in your home directory, but that has a small quota and you can easily use it all up. There is no quota on /gm2/app, but it is not backed up.

[4] If this directory does not exist, you can make it with the mkdir command.

```
$ mkdir /gm2/app/users/lyon/first-try  # On gm2gpvm
$ cd /gm2/app/users/lyon/first-try
```

If you are on your Mac, or some other machine, make the directory where you have room. Here's a script that will run on both Mac and gm2gpvm.

```
$ if [ -r /gm2/app/users/$USER ]; then  # Does /gm2/app/users/YOU exist?
$   # It does, let's use /gm2/app/users/$USER/first-try followed by random letters for uniqueness
$   TMPDIR=`mktemp -d /gm2/app/users/$USER/first-try.XXXX`
$ else
$     # We're not on gm2gpvm, let's just make a directory in your home area (hope there's room!)
```

```
$     TMPDIR=`mktemp -d ~/first-try.XXXX`
$ fi
$
$ # Change directory there
$ cd $TMPDIR
```

Note that all subsequent commands are the same for `gm2gpvm` and Mac or whatever.

Since you are starting out with a new area, you must choose a release. You should generally choose the latest, which will be specified in the output when you selected the release area. Just do what the command says,[5]

```
$ setup gm2 v5_00_00 -q e6:prof
```

[5] `setup` is a *ups* command. UPS is our release and product management system.

So here we are setting up g-2 release `v5_00_00` with the `e6:prof` qualifier. `e6` indicates the type of compiler we're using (in our case `gcc 4.9.1` with C++14 features turned on - this code is decided by the art team) and `prof` means we'll do a profile build. Profile builds are optimized and have debugging symbols turned on. We only use profile builds.

Now, you must create the development area. You will start using the `mrb` commands. *mrb* means "multi-repository build system" and is a build system used by Muon $g-2$, the art developers, and LBNF. You can get a list of `mrb` commands with (you don't have to type in the full path that you see below),

```
$ mrb -h
```

```
Usage /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/mrb/v1_03_00_gm2/bin/mrb  [-h for help]"

  Tools ( for help on tool, do "/cvmfs/oasis.opensciencegrid.org/gm2/prod/external/mrb/v1_03_00_gm2/bin/mrb <tool> -h" )

    newDev (n)                  Start a new development area
    gitCheckout (g)             Clone a git repository
    svnCheckout (svn)           Checkout from a svn repository
    setEnv (s)                  Setup development environment (mrbSetEnv)
    build (b)                   Run buildtool
    install (i)                 Run buildtool with install
    test (t)                    Run buildtool with tests
    setup_local_products (slp)  Setup local products (mrbslp) [not local sources]
    zapBuild (z)                Delete everything in your build area
    newProduct (p)              Create a new product from scratch
    changelog (c)               Display a changelog for a package
    bumpVersion (bv)            Bump version number of a package
    updateDeps (ud)             Update dependencies in CMakeLists.txt and product_deps
    updateCM (uc)               Update the master CMakeLists.txt file
    makeDeps (md)               Build or update a header level dependency list
    checkDeps (cd)              Check for missing build packages
    pullDeps (pd)               Pull missing build packages into MRB_SOURCE

  Aliases ( we use aliases for these commands because they must be sourced )

   mrbsetenv                   Setup a development enviornment and local products  [use this more often]
                               (source $MRB_DIR/bin/mrbSetEnv)

   mrbslp                      Setup only the products installed in the working localProducts_XXX directory
                               (source $MRB_DIR/bin/setup_local_products)
```

The `mrb` commands are the same if you are on `gm2gpvm` or your Mac.

To initialize your development area, do this in an empty directory.

```
$ mrb newDev
```

```
building development area for gm2 v5_00_00 -q e6:prof

MRB_BUILDDIR is /Users/lyon/first-try.NV8p/build_d13.x86_64
MRB_SOURCE is /Users/lyon/first-try.NV8p/srcs
INFO: cannot find releaseDB/base_dependency_database
      mrb checkDeps and pullDeps may not have complete information
MRB_PROEJCT IS gm2

IMPORTANT: You must type
    source /Users/lyon/first-try.NV8p/localProducts_gm2_v5_00_00_e6_prof/setup
NOW and whenever you log in
```

Read the output carefully. Some things to note:

- A build directory is created and note its name contains the flavor of your machine.[6] You can get to that directory easily with `cd $MRB_BUILDDIR` .
- A source directory is created for your source code. You can get to it easily by doing `cd $MRB_SOURCE` .
- You can ignore the message about the release database. That's a LBNF thing we don't use.
- The important message is indeed important. There is a set up script that you need to run that sets up your environment. Run that script now and whenever you log in to restore your development environment. You don't need to type in the whole path, since you are at the top of your development area.

[6] Mac is d13 (for Darwin version 13) and slf5, slf6 are marked as appropriate.

```
$ source localProducts_gm2_v5_00_00_e6_prof/setup
```

```
MRB_PROJECT=gm2
MRB_PROJECT_VERSION=v5_00_00
MRB_QUALS=e6:prof
MRB_TOP=/Users/lyon/first-try.NV8p
MRB_SOURCE=/Users/lyon/first-try.NV8p/srcs
MRB_BUILDDIR=/Users/lyon/first-try.NV8p/build_d13.x86_64
MRB_INSTALL=/Users/lyon/first-try.NV8p/localProducts_gm2_v5_00_00_e6_prof

PRODUCTS=/Users/lyon/first-try.NV8p/localProducts_gm2_v5_00_00_e6_prof:/cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2:/Users/lyon/Development/g-2/docs/localProducts_gm2_v5_00_00_e6_prof:/cvmfs/oasis.opensc
```

- A local products area is also created. This is your own personal release area that overlays the official one (so stuff you have in your personal release area override products in the official one).

3.3 Checkout code

Now you need to checkout some code. For this example, we'll use the `gm2artexamples` product. All of our code lives in `git` repositories on http://redmine.fnal.gov . The `mrb gitcheckout` command is used to

clone the git repositories (this is a convenience command so you don't have to remember the git URLs and other set up tasks).[7] Let's check out the `gm2artexamples` product. You must be in the `srcs` directory of your development area. The command is rather chatty.

[7] You can type `mrb g` for short.

```
$ cd srcs
$ mrb g gm2artexamples
```

```
git clone: clone gm2artexamples at /Users/lyon/first-try.NV8p/srcs
NOTICE: Running git clone ssh://p-gm2artexamples@cdcvs.fnal.gov/cvs/projects/gm2artexamples
Cloning into 'gm2artexamples'...
X11 forwarding request failed on channel 0
ready to run git flow init for gm2artexamples
Already on 'master'
Your branch is up-to-date with 'origin/master'.
Using default branch names.
Already on 'develop'
Your branch is up-to-date with 'origin/develop'.
Branch develop set up to track remote branch develop from origin.
X11 forwarding request failed on channel 0
Already up-to-date.
NOTICE: Adding gm2artexamples to CMakeLists.txt file
NOTICE: You can now 'cd gm2artexamples'

You are now on the develop branch (check with 'git branch')
To make a new feature, do 'git flow feature start <featureName>'
```

At this moment, you need to switch to a particular feature branch that is compatible with `gm2 v5_00_00`. Do the following,[8]

[8] This step will disappear shortly.

```
$ cd gm2artexamples
$ git flow feature track gm2_5
$ cd ..
```

```
X11 forwarding request failed on channel 0
Switched to a new branch 'feature/gm2_5'
Branch feature/gm2_5 set up to track remote branch feature/gm2_5 from origin.

Summary of actions:
- A new remote tracking branch 'feature/gm2_5' was created
- You are now on branch 'feature/gm2_5'
```

If you have more code to checkout, then run more `mrb g` commands.

3.4 Building code

Now that your code is checked out, you need to build it. The first step you need to do is to "extend" your environment with any products your build depends upon set up. The way to do this is to do `source mrb setEnv`.[9] You need `source` (or `.` for short) because your shell environment needs to be extended with new environment variables. You need to run this command after you log back into and start developing. If you do not make major changes to your code (you don't introduce new dependencies), then you only need to run the command once before you build.

[9] There are two shortcuts for `source mrb setenv`; you can do `. mrb s` or `mrbsetenv` (the latter is a bash function that does the `source` for you).

```
$ . mrb s
```

```
local product directory is /Users/lyon/first-try.NV8p/localProducts_gm2_v5_00_00_e6_prof
----------- this block should be empty ------------------
-------------------------------------------------------
ERROR: Cannot do unsetup, SETUP_CETPKGSUPPORT is not defined
The working build directory is /Users/lyon/first-try.NV8p/build_d13.x86_64
The source code directory is /Users/lyon/first-try.NV8p/srcs
----------- check this block for errors ----------------------
-----------------------------------------------------------------
```

For now, ignore the error about `SETUP_CETPKGSUPPORT` (it is benign). You should not see any errors between the dashed lines. If you do, then you have some product dependency mismatch (ask for help).

Now you can build your code. The build command is `mrb build`.[10]

[10] `mrb b` for short

```
$ mrb b
```

The long output is not shown. Hopefully there will be no compilation errors. If you get some, ask for help.

3.5 Testing

`gm2artexamples` is currently the only product that has unit tests. To try them, just do `mrb test`.[11]

[11] `mrb t` for short. A short build check will occur to ensure that everything is built.

```
$ mrb t
```

```
/Users/lyon/first-try.NV8p/build_d13.x86_64
calling buildtool -I /Users/lyon/first-try.NV8p/localProducts_gm2_v5_00_00_e6_prof -b -t
INFO: Install prefix = /Users/lyon/first-try.NV8p/localProducts_gm2_v5_00_00_e6_prof
INFO: CETPKG_TYPE = Prof

------------------------------------
INFO: Stage cmake.
------------------------------------

-- Product is gm2artexamples v2_00_00 e6:prof
-- Module path is /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/art/v1_12_02/Modules;/cvmfs/oasis.opensciencegrid.org/gm2/prod/external/cetbuildtools/v4_03_02/Modules
-- set_install_root: PACKAGE_TOP_DIRECTORY is /Users/lyon/first-try.NV8p/srcs/gm2artexamples
-- Building for Darwin d13 x86_64
-- set_install_root: PACKAGE_TOP_DIRECTORY is /Users/lyon/first-try.NV8p/srcs/gm2artexamples
-- Selected diagnostics option CAUTIOUS
-- cmake build type set to Prof in directory <top> and below
--    DEFINE (-D): ;NDEBUG
-- compiler flags for directory <top> and below
--    C++     FLAGS: -O3 -g -gdwarf-2 -fno-omit-frame-pointer -Werror -pedantic -std=c++1y -Wall -Werror=return-type
--    C       FLAGS: -O3 -g -gdwarf-2 -fno-omit-frame-pointer -Werror -pedantic   -Wall -Werror=return-type
-- Boost version: 1.56.0
-- Found the following Boost libraries:
--   chrono
--   date_time
--   filesystem
--   graph
--   iostreams
--   locale
--   prg_exec_monitor
--   program_options
--   random
--   regex
--   serialization
--   signals
--   system
--   thread
--   timer
--   unit_test_framework
--   wave
--   wserialization
-- CPACK_PACKAGE_NAME and CPACK_SYSTEM_NAME are gm2artexamples d13-x86_64-e6-prof
-- Configuring done
CMake Warning (dev):
  Policy CMP0042 is not set: MACOSX_RPATH is enabled by default.  Run "cmake
  --help-policy CMP0042" for policy details.  Use the cmake_policy command to
  set the policy and suppress this warning.

  MACOSX_RPATH is not specified for the following targets:

   gm2artexamples_DataObjects_dict
   gm2artexamples_DataObjects_map
   gm2artexamples_HitAndTrackObjects_dict
   gm2artexamples_HitAndTrackObjects_map
```

```
   gm2artexamples_Lesson1_HelloWorld1_module
   gm2artexamples_Lesson1_HelloWorld2_module
   gm2artexamples_Lesson1_MyDatumReader_module
   gm2artexamples_Lesson1_ProduceMyLittleDatum_module
   gm2artexamples_Lesson2_makeHits_module
   gm2artexamples_Lesson2_makeRotatedHits_module
   gm2artexamples_Lesson2_makeSimpleTracksFromNewHits_module
   gm2artexamples_Lesson2_makeSimpleTracksFromOldHits_module
   gm2artexamples_Lesson2_readHits_module
   gm2artexamples_Lesson2_readSimpleTracks_module
   test_MyLittleDatumAnalyzer_module
   test_MyLittleDatumProducer_module

This warning is for project developers.  Use -Wno-dev to suppress it.

-- Generating done
-- Build files have been written to: /Users/lyon/first-try.NV8p/build_d13.x86_64

------------------------------------
INFO: Stage cmake successful.
------------------------------------


------------------------------------
INFO: gm2artexamples version 2.00.00 configured.
------------------------------------


------------------------------------
INFO: Stage build.
------------------------------------

[  3%] Built target gm2artexamples_DataObjects
[  9%] Built target gm2artexamples_DataObjects_dict
[ 15%] Built target gm2artexamples_DataObjects_map
[ 21%] Built target gm2artexamples_HitAndTrackObjects
[ 28%] Built target gm2artexamples_HitAndTrackObjects_dict
[ 34%] Built target gm2artexamples_HitAndTrackObjects_map
[ 37%] Built target gm2artexamples_Lesson1_HelloWorld1_module
[ 40%] Built target gm2artexamples_Lesson1_HelloWorld2_module
[ 43%] Built target gm2artexamples_Lesson1_MyDatumReader_module
[ 46%] Built target gm2artexamples_Lesson1_ProduceMyLittleDatum_module
[ 50%] Built target gm2artexamples_Lesson2_makeHits_module
[ 53%] Built target gm2artexamples_Lesson2_makeRotatedHits_module
[ 56%] Built target gm2artexamples_Lesson2_makeSimpleTracksFromNewHits_module
[ 59%] Built target gm2artexamples_Lesson2_makeSimpleTracksFromOldHits_module
[ 62%] Built target gm2artexamples_Lesson2_readHits_module
[ 65%] Built target gm2artexamples_Lesson2_readSimpleTracks_module
[ 68%] Built target +Users+lyon+first-try.NV8p+build_d13.x86_64+gm2artexamples+bin+myLittleDatum_wr.sh
[ 71%] Built target +Users+lyon+first-try.NV8p+build_d13.x86_64+gm2artexamples+bin+very_simple_test.sh
[ 75%] Built target +Users+lyon+first-try.NV8p+build_d13.x86_64+gm2artexamples+test+MyLittleDatum_test.d+MyLittleDatum_test.fcl
[ 78%] Built target +Users+lyon+first-try.NV8p+build_d13.x86_64+gm2artexamples+test+MyLittleDatum_test.d+messageDefaults.fcl
[ 81%] Built target +Users+lyon+first-try.NV8p+build_d13.x86_64+gm2artexamples+test+myLittleDatum_wr.sh.d+MyLittleDatum_r.fcl
[ 84%] Built target +Users+lyon+first-try.NV8p+build_d13.x86_64+gm2artexamples+test+myLittleDatum_wr.sh.d+MyLittleDatum_w.fcl
[ 87%] Built target +Users+lyon+first-try.NV8p+build_d13.x86_64+gm2artexamples+test+myLittleDatum_wr.sh.d+messageDefaults.fcl
[ 90%] Built target simple_test
[ 93%] Built target test_MyLittleDatumAnalyzer_module
[ 96%] Built target test_MyLittleDatumProducer_module
[100%] Built target test_with_boost

real    0m4.623s
user    0m1.538s
sys 0m1.187s

------------------------------------
INFO: Stage build successful.
------------------------------------


------------------------------------
INFO: Stage test.
------------------------------------

Test project /Users/lyon/first-try.NV8p/build_d13.x86_64
    Start 1: very_simple_test.sh
1/5 Test #1: very_simple_test.sh .............   Passed    0.01 sec
    Start 2: simple_test
2/5 Test #2: simple_test .....................   Passed    0.01 sec
    Start 3: test_with_boost
3/5 Test #3: test_with_boost .................   Passed    0.02 sec
    Start 4: MyLittleDatum_test
4/5 Test #4: MyLittleDatum_test ..............   Passed    0.58 sec
    Start 5: myLittleDatum_wr.sh
5/5 Test #5: myLittleDatum_wr.sh .............   Passed    0.80 sec

100% tests passed, 0 tests failed out of 5

Total Test time (real) =   1.43 sec

------------------------------------
INFO: Stage test successful.
------------------------------------
```

3.6 Running

There are several fcl files you can run for `gm2artexamples`.

```
$ ls $MRB_SOURCE/gm2artexamples/fcl
```

```
CMakeLists.txt
hello1.fcl
hello2.fcl
makeAndReadDatum.fcl
makeAndReadTracksFromOldHits.fcl
makeDatum.fcl
makeHits.fcl
makeHitsRotated.fcl
makeTracksFromNewHits.fcl
makeTracksFromOldHits.fcl
messageservice.fcl
minimalMessageService.fcl
readDatum.fcl
readHits.fcl
readSimpleTracks.fcl
```

Our `art` executable is called `gm2`. FCL files are found by the `$FHICL_FILE_PATH` search path.

```
$ gm2 -c hello1.fcl
```

```
%MSG-i MF_INIT_OK:  17-Dec-2014 02:01:13 CST JobSetup
Messagelogger initialization complete.
%MSG
Begin processing the 1st record. run: 1 subRun: 0 event: 1 at 17-Dec-2014 02:01:13 CST
Hello, world. From analyze. run: 1 subRun: 0 event: 1
Begin processing the 2nd record. run: 1 subRun: 0 event: 2 at 17-Dec-2014 02:01:13 CST
Hello, world. From analyze. run: 1 subRun: 0 event: 2

TrigReport ---------- Event  Summary ------------
TrigReport Events total = 2 passed = 2 failed = 0

TrigReport ------ Modules in End-Path: end_path ------------
TrigReport  Trig Bit#    Visited     Passed     Failed      Error Name
TrigReport     0    0          2          2          0          0 hello

TimeReport ---------- Time  Summary ---[sec]----
TimeReport CPU = 0.000063 Real = 0.000098

Art has completed and will exit with status 0.
```

3.7 Logging in again

At some point, you will want to log out of your machine and log back in later to continue your work. To reconstitute your development environment, you need to,

- Select the release area

  ```
  source /grid/fermiapp/gm2/setup # on gm2gpvm
  source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # On Mac
  ```

- Change directory to your development area

  ```
  cd ~/Development/g-2/first-time  # On my Mac
  ```

- Run the setup script in local products (this will re-select the chosen g-2 release)

```
source localProducts_gm2_v5_00_00_e6_prof/setup
```

- Extend the environment for the products your build depends upon (don't forget the leading dot)

```
. mrb s
```

Now you are set to build (`mrb b`), run (`gm2 -c FCL_FILE`), and develop.

3.8 Summary

Here is a summary of the commands for `gm2 v5_00_00`.

3.8.1 To checkout, build and run *gm2artexmples* to a new development area

```
# Log into machine (e.g. gm2gpvm.fnal.gov)

# Select release area
source /grid/fermiapp/gm2/setup  # On gm2gpvm
source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # On Mac

# Create development area
mkdir /gm2/app/users/lyon/first-time # For me on gm2gpvm
mkdir ~/Development/g-2/first-time   # For me on my Mac
cd <THAT_DIRECTORY>

# Setup the release
setup gm2 v5_00_00 -q e6:prof

# Initialize Development area
mrb newDev
source localProducts_gm2_v5_00_00_e6_prof/setup

# Checkout code
cd srcs
mrb g gm2artexamples

# Get right branch (for now)
cd gm2artexamples
git flow feature track gm2_5
cd ..

# Extend environment with build dependencies
```

```
. mrb s

# Build it
mrb b

# Test it
mrb t

# Run it
gm2 -c hello1.fcl
```

3.8.2 Restoring environment when logging in again later

Here's what you do to restore your environment

```
# Log into machine (e.g. gm2gpvm.fnal.gov)

# Select release area
source /grid/fermiapp/gm2/setup  # On gm2gpvm
source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # On Mac

# cd to development area
cd /gm2/app/users/lyon/first-time # For me on gm2gpvm
cd ~/Development/g-2/first-time   # For me on my Mac

# Restore basic environment
source localProducts_gm2_v5_00_00_e6_prof/setup

# Extend environment with build dependencies
. mrb s

# Now you can work!! For example
mrb b  # Build it if you've made a change since last time
mrb t  # Test it
gm2 -c hello1.fcl # Run it
```

4
Developer Workflow

The steps you follow to develop code is described here (eventually).

5
Using a Mac for Development

Our code builds on a Mac[1], so you can use your Mac laptop or desktop to develop code. Note that you should use a linux machine and the Grid to do large runs. While art and our code works on a Mac, you should still consider it experimental. While development can occur on a Mac, official analyses should be performed on Scientific Linux.

One of the advantages of development on a Mac with OSX the very nice Xcode Integrated Development Environment (IDE). Here are some things you can do that with Xcode that are hard to do from the command line or with a source code editor like emacs or vi.

- A very convenient code browser with forward and backward buttons
- Click on an include file line and jump to that file
- Click on a class or object name and jump to its definition
- Xcode understands projects - that is all of the source files that go together in a development area
- Easily search for text and objects in all files within the project
- Build the code and if there's an error, jump to the offending line with the error displayed there
- Code completion and snippets
- Debugging within the IDE (super nice on Xcode!)
- Built in git management (you can commit and push to Redmine from within Xcode), easy to read file comparisons, quick "blame" to see who changed lines

Only some of the most important features of Xcode will be explained here. You can learn lots more about Xcode from its help system.

[1] At this time, only OSX 9 (Mavericks) works with art and our code. Fermilab does not allow OSX 10 (Yosemite) installs at the lab as that OS has not been approved for use. Hopefully Yosemite will be allowed soon.

5.1 Installing CVMFS

We distribute our code via the CERN Virtual Machine File System (CVMFS). It allows you to "subscribe" to the published directories. You will get updates automatically as they are pushed out from Fermilab. We use the Open Science Grid's OASIS server. See https://cdcvs.fnal.gov/redmine/projects/g-2/wiki/InstallingOasisOnMacLaptop for how to install and configure CVMFS on your Mac. Once you install, configure and mount CVMFS, you will find our releases at `/cvmfs/oasis.opensciencegrid.org/gm2/prod` . See elsewhere in this manual for how to set up a development area, checkout code and build with the `mrb` commands from the command line. Those commands all work on the Mac.

5.2 Installing Xcode

Get Xcode from the `App Store` application on your Mac. Xcode is a free product.

Once you install it, you may need to install the Command Line Tools. Start up Xcode by clicking on its icon,[2] go to the menu option Xcode → Preferences, and on the box that appears select the Locations tab. See Fig. 5.1. If the command line tools say install, or something similar, then click on the appropriate button to download these tools. You may have trouble building g-2 code if you do not follow this step, as necessary system headers are downloaded with the command line tools.

[2] As you will read below, you will in general not start Xcode this way, but for this step clicking on the icon is fine.

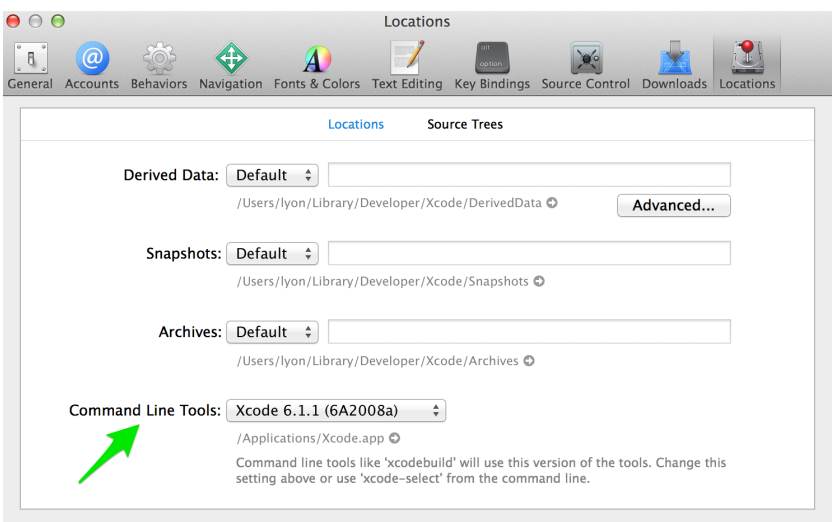

Figure 5.1: Command Line Tools

5.3 Running Xcode

It is easiest to start with code you are already working on. Create a development area on your Mac, checkout your code[3], and get everything set up[4]. You should then do the `cmake` part of the build step so that `cmake` writes its cache files to the build area (if you let Xcode this, it can go wrong). That is, do `mrb b -C`.

[3] or go to a directory you are already working on

[4] e.g. up to and including `. mrb s`

When you run Xcode, you want Xcode to be executing within this already configured shell environment. To do this, do **not** run Xcode by clicking on its icon. You must instead run it from the command line from the same terminal window you are using to work on your project. The command to run is (assuming you installed Xcode in the usual place in `/Applications`),

```
$ /Applications/Xcode.app/Contents/MacOS/Xcode &
```

Note that on my Mac, Xcode will start but its window will not appear in front. Click on the Xcode icon in your dock or change the application to it with `Command-tab`.

You may see some warnings and error messages appear in your terminal now and occasionally later. You may even see messages telling you to file a bug report with Apple. I ignore all of these messages as they all seem benign.[5]

[5] If you run Xcode by clicking its icon, then these messages go to your system log. But we need to run Xcode from the command line so that it is running within the configured shell environment (e.g. so `mrb b` will work).

A longer example, restarting work in a development area, is as follows,

```
$ source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup
$ cd ~/Development/g-2/myAwesomeSimulationDevArea  # replace with your area
$ source localProducts*/setup
$ . mrb s   # Set up development environment
$ mrb b -C  # Run cmake step of build
$ /Applications/Xcode.app/Contents/MacOS/Xcode &  # Start Xcode
```

5.4 Configure Xcode for your project

This section will teach you how to start a new Xcode project. You need to do this if you haven't configured Xcode for your development area. There's a lot to do, but it's all pretty straightforward. Your reward will be working within a really nice IDE that will do a lot of work for you. So read on and follow the directions below.

1. Start up Xcode following the instructions of the previous section. If you haven't started Xcode for awhile, or are starting it without having a project set up, you should see the screen shown in Fig. 5.2.[6] Click on `Create a new Xcode Project`.

[6] If you do not see this welcome window or an already in-progress project appears, then you can start a new one by selecting File → Close Project and then File → New → Project...

Figure 5.2: Xcode Welcome Screen

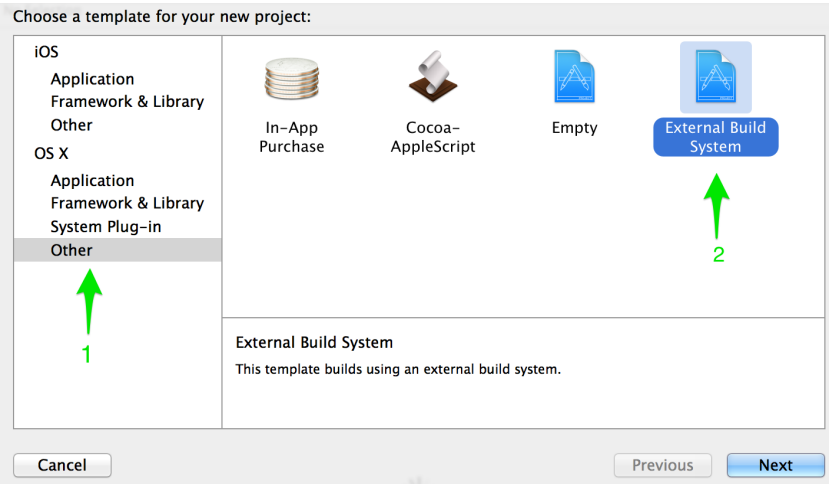

Figure 5.3: Xcode's new project window

2. The new project window will appear, as shown in Fig. 5.3. Select "OS X → Other" and then click on the "External Build System" icon. In the next window that appears, fill in the name of your project (I keep it the same name as the development area, but you can choose anything you want). For the build tool, enter `make`. You can fill in anything for the other input lines. Click `Next`.

3. Xcode will now ask you to choose a directory within which to store the project information. I have a special `Xcode` directory **not** in my g-2 development area for these files. I suggest you do the same, as we don't want to pollute our source code with Xcode poop. For example, my development area will be in `~/Development/g-2/myAwesomeSimulationDevArea` and I'll tell Xcode to put its project files in `~/Development/Xcode`. Do this and click on `Create`.

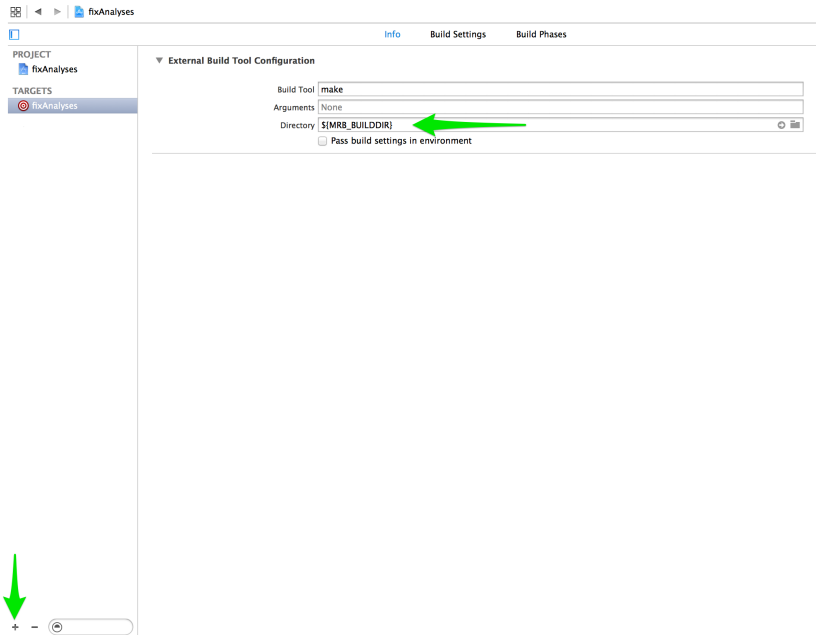

Figure 5.4: Main project info screen

4. You are now at your main project screen showing the Build configuration as seen in Fig. 5.4. Under "External Build Tool Configuration → Arguments" replace `$(ACTION)` with blank (empty). By "Directory" enter `${MRB_BUILDDIR}`. "We also want to set up a documentation index so you can easily search around the project. To do this, we need to add another target. Click on the `+` symbol on the bottom of the screen as shown in the figure. On the new window that appears, select"OS X → Framework & Library" and

then click on the "Library" icon. Press "Next". For the product name, enter the name of your project with `-docs` at the end (e.g. `myAwesomeSimulation-docs`). For the "Framework", be sure that "None (Plain C/C++ Library)" is selected. Leave the rest alone and press "Finish". A "Products" folder will appear on the left in the file chooser. You can select it and press Delete to remove it, as it is not needed.

5. You should now have a screen where your new `-docs` target is selected under "TARGETS" and the "Build Settings", "All", and "Combined" are shown selected. You must now enter the header search paths that will be used for indexing. To make this easy, we'll insert a some simple marker text into the header search list and then write a little script on the command line to fill in the actual paths. Here's what you need to do:

6. There's a little search box underneath "Build Rules"; in there type "header" and press Return. Now look on the screen for "Search Paths" and underneath that "Header Search Paths". If the little triangle is pointing downward showing "Debug" and "Release", click on the triangle to close it. Now **double** click on the *contents* of the "Header Search Path" (the text probably starting with "/Applications/Xcode.app"). A box will appear. Click on the "+" and add `NNN` and press Return. Click on the "+" again and add `${MRB_SOURCE}` and press Return. Now click somewhere outside of the box to close it.

7. Go to your terminal window (the same one that you started Xcode from) and type in the following, replacing `<project_dir>` with the directory where you placed the project and `<project_name>` with the name of the project. For me, that would be `~/Development/g-2/xcode/myAwesomeSimulation` (yes, include the project name) and `myAwesomeSimulation` respectively. You should probably cut and paste.

```
# Search typical include paths
$ r=$(env | grep INC | \
      egrep -i 'art|g4|cet|message|fhicl|clhep|root_' | grep -v DIR | \
      awk 'BEGIN { FS="="; ORS=" "} ; { print "\"${"$1"}\"," } ')
$ sed -i '.bak' "s/NNN,/$r/" <project_dir>/<project_name>.xcodeproj/project.pbxproj
```

The lines above are for the typical include directories that you will be using. If you don't use anything from Geant, you can leave out the `g4` item in the grep list. If, for some reason, you want to search **every** include area you are building against (indexing will be much slower), then do,

```
# Search all include paths (indexing will be slower than for the typical list)
$ r=$(env | grep INC | awk 'BEGIN { FS="="; ORS=" "} ; { print "\"${"$1"}\"," } ')
$ sed -i '.bak' "s/NNN,/$r/" <project_dir>/<project_name>.xcodeproj/project.pbxproj
```

8. If all went well, then your Xcode screen may go mostly blank. Just double click on the project name in the top of the left strip to bring back the build information. If you see a parse error, then something went wrong. The `sed` command made a backup file of the original file contents. Copy that old file back with `pushd <project_dir>/<project_name>.xcodeproj ; cp project.pbxproj.bak project.pbxproj ; popd` and try again. If things worked, then you can delete the `project.pbxproj.bak` file.

9. You are almost ready. Now you have to add your source files to the Xcode project. Select the menu item File → Add Files to ... In the dialog box that appears, first, by "Add to targets" check both the regular target and the "-docs" target. *Make sure the box by the docs target is checked.* If you don't check that box, the files will not be indexed.[7] Now navigate the in the upper part of the box to your development area's `srcs` directory. Double click on it to select it. The directory will appear in the file list on the left strip.

10. In the activity box (upper center of the main window) you should eventually see a message that Xcode is indexing. This may take several minutes to finish.

[7] If you forget this step, right click on `srcs` in the file list on the left strip and select "Delete". Then click on "Remove References". Then follow the steps again, correctly this time.

Your Xcode is now configured for your project. If you quick Xcode and start it again, you can re-select the project from the welcome window, or start another new project.

5.5 Things you can do with Xcode

Xcode is a program with enormous capabilities. Because we do not use Apple's compiler, many Xcode functions won't work (like its intelligent code sensing). But a lot does work that is very convenient.

5.5.1 Navigating source code

The file list on the left strip of the Xcode window allows you to easily select files to view and edit. Right clicking on a file bring up many options, including the ability to check a file into git.

Within the source code, you can Command-click on an include file name, or an object and if it is indexed, Xcode will jump to the file and the appropriate line. This capability is very powerful and it makes understanding code much easier than it would be otherwise. At the top of the editor are forward and backward navigation buttons. If a

Command-click takes you to a different file, pressing the back button will bring you back to the original file.

Right clicking on a line and then selecting "Show blame for line" will display the git commit that wrote or changed that line last. Very handy if you want to understand who made a particular change.

Xcode is not a tabbed editor, but is has the notion of an "Assistant editor" that can show you another file in the main Xcode window. To specify the default location of the Assistant editor, use the menu item "View" → "Assistant Editor" and then where you want the editor to appear. To make the editor appear, click on the Tuexedo icon towards the top right of the Xcode window. You can configure the editor to show you a particular file, or show you the counterpart file to the one in the main editor (e.g. the `.cc` file if you are looking at the header, or vice-versa). Note that many of the Assistant Editor selections (Superclass, Subclass) will not work because we are not using the Apple compiler.

5.5.2 Integration with git

Xcode is very heavily integrated with git. You will notice that in the file strip of the left side of the window, files that you have modified or added will be indicated with an "M" or "A" respectively. The "Source Control" menu option has many functions such as Check in and Push/Pull. Push/Pull will automatically work against the Redmine server.

Xcode's file comparison display is great. Click on the wave icon (to the right of the tuxedo) towards the upper right of the Xcode window. You can compare your current file with what is in git, in a branch, etc.

Merging in Xcode is very convenient as if there are merge conflicts, the version comparison window will help you sort them out.

5.5.3 Building your code

Since you are running Xcode in your development, and since you have configured Xcode to use `mrb`, things will just work. If you want to do a build, choose the menu item "Product" → "Build" or press Command-b. The acivity window will indicate that a build has begun. To check on the progresss of the build, in the file strip on the left side, click on the right most icon on the top bar (looks like a cartoon bubble). You will then see all of the builds you have performed. Clcik on the top-most one to see the current build. Unfortunately, Xcode shows some strange subset of the output of the build. To see everything, at the top of the output (you may need to scroll up), click on the line that says "Run external build tool". In fact click on the triangle to hide the text below. On the right side of the line, a little lined icon will appear.

Click on that and the full output will be displayed. Unfortunately, you cannot search the output.

If there was a build failure, you will see red explanation marks in Various Xcode windows. Click on them and you will be taken to the file with the first error, which a nice indicator showing where and what the error regards.

Once builds are sucessful, you should run `gm2`. You must do that from the command line in the terminal that you used to start Xcod and not from within Xcode itself.

5.5.4 Debugging

Debugging from a command prompt is less than fun. The Xcode IDE makes debugging a nice experience. More goes here.

5.5.5 Zapping your build area or changing the `srcs/CmakeLists.txt` file

There are instances when you need to re-establish your development environment. Two of them are when you zap your build area or when you change the `srcs/CMakeLists.txt` file. When you do the latter, you are telling the system to stop building some packages or to start building some pacakges you haven't built before. In that case, you need to zap the build area and re-establish the developement environment. You need to quit and re-start Xcode so that Xcode is running in the correct environment. So, do the following,

```
# Quit Xcode
mrb z     # zap
. mrb s   # Setup development environment (the zap removes this)
mrb b -C  # cmake step so that Xcode works
$ /Applications/Xcode.app/Contents/MacOS/Xcode &  # Start Xcode again
```

6
Getting Started with the Simulation

This section gives information and instructions on how to get started with the Muon g-2 simulation. If you are brand new to the simulation, then you have several things you need to learn,

- Geant4
- ArtG4
- Gm2RingSim and its associated packages

Let's go through these things one at a time.

6.1 Geant4

`Geant4` is a toolkit for the simulation of particles passing through matter and fields. You can create all manner of apparatuses, shoot particles at it, and see what the particles will do. `Geant4` has extensive physics models that can handle a wide variety of situations. We use `Geant4` to build our Muon g-2 ring with its detectors and then shoot muons into it. `Geant4` figures out how those muons will behave. It can do decays, spin tracking, interactions with the calorimeter crystals and optical photons, etc. It is quite an extensive package. The home page for `Geant4` is at http://geant4.cern.ch/ . There are three main parts of `Geant4`,

Building the apparatus and detectors You must define the shapes and materials that the particles will be passing through. `Geant4` has an extensive library of materials and shapes to choose from. You must also create "sensitive detectors", which are parts of the apparatus where geant will record interactions and energy loss as hits.

Defining "actions" Geant processes the simulation in many steps, including starting and ending events, tracking new particles, and stepping through parts of the simulation. You can add your code in these processes via `actions`.

Examining hits The ultimate goal of the simulation is to record interactions of particles with the sensitive detectors in the apparatus. Such information is the "truth". You must then have code outside of geant that determines the response of the detectors to these hits (usually called the digitization step).

On the Geant home page are various user guides. The best one to look at for a newcomer is the Users's Guide for Application Developers. Part of learning geant is going through the extensive set of examples. Fortunately, we have all of the examples distributed in our g-2 release.

6.1.1 Building and running the Geant4 examples

See section 3.1 and follow those instructions to set up your environment. Note that `Geant4` has its own build system for the examples, so we will not be using the usual g-2 development area. First, we need to chose a release area.

bash $ if [-r /grid/fermiapp/gm2/setup]; then # Does /grid/fermiapp/gm2/setup exist? $ source /grid/fermiapp/gm2/setup # We're on gm2gpvm $ else $ source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # We're on a Mac $ fi $ $ $ $ $ We will only set up `geant4` along with the `cmake` build system, which is all we need to run the examples (below are the latest versions of geant and cmake we have in the release), $

```
$ setup geant4 v4_9_6_p03e -q e6:prof
$ setup cmake v3_0_1
```

```
/tmp/a: line 2: setup: command not found
/tmp/a: line 3: setup: command not found
```

Let's make a directory to do some work in.

```
$ if [ -r /gm2/app/users/$USER ]; then  # Does /gm2/app/users/YOU exist?
$   # It does, let's use /gm2/app/users/$USER/first-try followed by random letters for uniqueness
$   TMPDIR=`mktemp -d /gm2/app/users/$USER/geant-ex.XXXX`
$ else
$     # We're not on gm2gpvm, let's just make a directory in your home area (hope there's room!)
$     TMPDIR=`mktemp -d ~/geant-ex.XXXX`
$ fi
$
$ # Change directory there
$ cd $TMPDIR
```

You can find the geant examples at, `$GEANT4_DIR/source/geant4.9.6.p03/examples`,

```
$ ls $GEANT4_DIR/source/geant4.9.6.p03/examples
```

```
ls: /source/geant4.9.6.p03/examples: No such file or directory
```

The `README` file describes the different examples. The build instructions here are based on the `README.HowToRun` file. See that file for more information.

Let's try to build and run the `N05` example in the `novice` directory.

First, we need to make a build area,

```
$ mkdir n05-build
$ cd n05-build
```

Now we run `cmake`[1] with some parameters to set up the build system.

[1] On the Mac, you may see a message about using the AppleClang C compiler. That is not a problem because Geant is all C++ and so the C compiler will not be used.

```
$ export CMAKE_PREFIX_PATH=$GEANT4_FQ_DIR
$
$ cmake -DCMAKE_BUILD_TYPE=Debug \
$       -DCMAKE_CXX_COMPILER=$GCC_FQ_DIR/bin/g++ \
$       -DCMAKE_CXX_FLAGS="-std=c++1y" \
$       $GEANT4_DIR/source/geant4.9.6.p03/examples/novice/N05
```

```
CMake Error: The source directory "/source/geant4.9.6.p03/examples/novice/N05" does not exist.
Specify --help for usage, or press the help button on the CMake GUI.
```

And now we run `make`,

```
$ make
```

```
make[3]: *** No targets specified and no makefile found.  Stop.
```

There will now be an executable `example05` in the build directory. All of the necessary files you need to run (`.in`, `.mac`, `.gdml`) will also be copied to the build directory. Files with `.in` are input macro files. You can run them with (for this example),

```
$ ./example05 example05.in > out   # There is lots of output, so redirect
$ less out  # Examine the output
```

You can also run in interactive mode. This mode will allow you to see the visualizations. For example,

```
$ ./example05

# You will now be at the Idle> prompt
# First, let's run "vis.mac" to set up visualization
Idle> control/execute vis.mac

# Now run some commands.  Best to have another window so you can
# look at example05.in for hints
```

```
Idle> /gun/particle e-
Idle> /gun/energy 1 GeV
Idle> /gun/position 0 0 0
Idle> /gun/direction 0 .6 1.
Idle> /run/beamOn 1

# You will see one particle shot into the apparatus
# You can end with exit
Idle> exit
```

Be sure to look at the code and understand what it is doing.

If you want to try a different example, use its directory on the last line of the call to `camke` above.

If you want to alter the example code, then you will have to copy the source code directory to your own directory. Build it the same way as above, but with the last line of the `cmake` call pointing to your source directory.

7
Running the simulation

This section gives you very brief instructions on how to build and run the `gm2ringsim` simulation. More details will be coming in future versions of this document.

Be sure you are familiar with the basics in section 3.

At this moment, the released code for the simulation is quite old,[1] so you will have to download and build all the simulation code yourself. The first build will take awhile (about 20 minutes on the `gm2gpvm` machines), but subsequent builds will be faster.

Set up the `gm2 v5_00_00` release and create a new development area. Obtain the source code and switch to the `gm2 v5_00_00` branch by following the instructions below.[2] Perform the following steps in your development area, which we assume is ready to go.

[1] This should be remedied soon as `gm2 v5_00_00` is adopted.

[2] Very soon, the branch switching step will be unnecessary once we adopt `gm2 v5_00_00`. And unfortunately, I was not consistent with how I named the branches. Sorry.

```
cd srcs

# Get artg4
mrb g artg4
cd artg4
git flow feature track gm2_v5
cd ..

# Get gm2geom
mrb g gm2geom
cd gm2geom
git flow feature track g5
cd ..

# Get gm2dataproducts
mrb g gm2dataproducts
cd gm2dataproducts
git flow feature track g5
cd ..
```

```
# Get gm2ringsim
mrb g gm2ringsim
cd gm2ringsim
git flow feature track g5
cd ..
```

You can now build the code with,

```
. mrb s
mrb b
```

There are many `fcl` files that you can use to run the simulation. Here's a list of some of them,

`BeamDiagnosticMuPlus.fcl` Shoot individual muons that go around the ring with a rudimentary particle gun with the fiber harp deployed.

`BeamDiagnosticMuPlusMuonGasGun.fcl` Simulation with fiber harp deployed using the gas gun. The gas gun makes muons randomly appear in the ring right before decay.Since geant does not track muons around the ring, this is a very fast simulation.

`ProductionMuPlus.fcl` Shoot individual muons that go around the ring with the ring in data taking state (e.g. no fiber harp).

`ProductionMuPlusMuonGasGun.fcl` Same as above, but using the muon gas gun. Very fast simulation.

`beamtransport_gun.fcl` Muons are not tracked around the ring. Instead, the position and momentum of the muon is calculated using the beam equations of motion and the muon appears in the ring just before it decays. A very accurate and fast simulation.

`inflector_gun.fcl` A very slow but accurate simulation of muons going through the inflector and around the ring.

8
Writing Source Code

Warning: This section needs to be reviewed and cleaned up.

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

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

[2] Examine `gm2ringsim` for more examples.

8.1 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.[3] The `mrb newProduct` command will create a skeleton file for you.

[3] 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

```
# Ensure we are using a moden version of CMake
CMAKE_MINIMUM_REQUIRED (VERSION 2.8)

# Project name - use all lowercase
PROJECT (gm2analyses)

# Define Module search path
set( CETBUILDTOOLS_VERSION $ENV{CETBUILDTOOLS_VERSION} )
if( NOT CETBUILDTOOLS_VERSION )
  message( FATAL_ERROR
         "ERROR: setup cetbuildtools to get the cmake modules" )
endif()
set( CMAKE_MODULE_PATH $ENV{CETBUILDTOOLS_DIR}/Modules
                                        ${CMAKE_MODULE_PATH} )

# art contains cmake modules that we use
set( ART_VERSION $ENV{ART_VERSION} )
if( NOT ART_VERSION )
  message( FATAL_ERROR
```

```
         "ERROR: setup art to get the cmake modules" )
endif()
set( CMAKE_MODULE_PATH $ENV{ART_DIR}/Modules
                                        ${CMAKE_MODULE_PATH} )

# Import the necessary macros
include(CetCMakeEnv)
include(BuildPlugins)
include(ArtMake)
include(FindUpsGeant4)

# Configure the cmake environment
cet_cmake_env()

# Set compiler flags
cet_set_compiler_flags( DIAGS VIGILANT WERROR
   EXTRA_FLAGS -pedantic
   EXTRA_CXX_FLAGS -std=c++11
)

cet_report_compiler_flags()

# Set include and library search paths (the version numbers
# are minimum -  if actual version of product is below specified,
# will get error)

# Everyone should include these
find_ups_product(cetbuildtools v3_07_08)
find_ups_product(art v1_08_10 )
find_ups_product(fhiclcpp v2_17_12)
find_ups_product(messagefacility v1_10_26)

# This project uses code from gm2ringsim,
# gm2dataproducts, and gm2geom
find_ups_product(gm2ringsim v1_00_00)
find_ups_product(gm2dataproducts v1_00_00)
find_ups_product(gm2geom v1_00_00)

# This project uses code from Root
find_ups_root(v5_34_12)

# Make sure we have gcc
cet_check_gcc()

# Macros for art_make and simple plugins (must go after
# find_ups lines)
include(ArtDictionary)

# Specify subdirectories to build
add_subdirectory( ups )  # Every project needs a ups subdirectory
add_subdirectory( DisplayDataProducts )
```

```
add_subdirectory( calo )
add_subdirectory( fcl )
add_subdirectory( test )
add_subdirectory( util )

# Packaging facility - required for deployment
include(UseCPack)
```

8.1.1 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.[4] 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.

[4] The `add_subdirectory` directory 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_ups_product` or similar line.[5]

[5] See section 8.8 for instructions.

8.2 Organizing Source Code

The build system we use is quite flexible and you can organize your code in many ways. You may be used to having all of your header files in an `include` directory with the `.cc` files in other directories. This artificial separation is unnecessary. You may group files together any way you like and may have header files and implementation files in the same directory. Typically, it is best to group files by topic or functionality.

8.3 Writing Modules

Modules are plugins to art that perform certain functions (analyzers, producers, filters, and output modules). See section 10 of the Art Work Book[6] for more information. Only reminders will be given here.

[6]

You should use `artmod` to write the skeleton of the module. Do `artmod --help-types` to see the list of module types it will make. Then just run it, giving the name of the class you want including any namespace specification. For example,

```
  artmod producer tracking:TrackFinder
  artmod analyzer gm2analysis::CalorimeterDiags
```

Remember that you specify the class name, not the file name (so do not give `_module` in the name).

8.4 Writing Services

TODO

8.5 Writing Input Source Modules

TODO

8.6 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. 8.1.1).[7] If your subdirectory has code to build, then the directory `CMakeLists.txt` file needs to have

```
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. 8.6.1) 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 `mrb install`. To do that, you need a line the directory `CMakeLists.txt` file with

```
install_headers( )  # No arguments
```

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),

```
# install all *.fcl files in this directory to the release area
file(GLOB fcl_files *.fcl)
install( FILES ${fcl_files}
         DESTINATION ${product}/${version}/fcl )

# Also install to the build area
foreach(aFile ${fcl_files})
  get_filename_component( basename ${aFile} NAME )
  configure_file(
```

[7] 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.

```
          ${aFile} ${CMAKE_BINARY_DIR}/${product}/fcl/${basename}
          COPYONLY )
endforeach(aFile)
```

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

8.6.1 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.[8] 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,

[8] See Sec. 8.8 for how to tell if you are using external code.

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

```
art_make(
      LIB_LIBRARIES
      library1
      library2   # if needed
)
```

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

```
art_make(
      MODULE_LIBRARIES
      library1
      library2   # if needed
  )
```

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

```
art_make(
      SERVICE_LIBRARIES
      library1
      library2   # if needed
  )
```

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

```
art_make(
      SOURCE_LIBRARIES
      library1
      library2   # if needed
  )
```

If you have a mixture of sources in your directory, you can string the calls together. For example,[9]

```
art_make (
     LIB_LIBRARIES
        ${ROOT_GPAD}
     MODULE_LIBRARIES
        gm2analyses_util
        gm2analyses_strawtracker_util
       )
```

[9] In the example to the left, regular sources get linked against Root's `libGpad.so` (see Sec. 8.8.2) and modules get linked against code built in the `srcs/gm2analyses/util` and `srcs/gm2analyses/strawtracker/util` directories (see Secs. 8.8.4 and 8.8.5).

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.

8.7 Libraries produced from building

Every directory in your project that has code to build generates at least one library.[10] Say, for example, you have a directory called `gm2analyses/calo`. Regular sources (not modules, services, nor input sources) get compiled and the objects go into a library called `libgm2analyses_calo.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_Calo_module.cc` then that code will go into a library called `libgm2analyses_calo_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.

[10] 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. 8.6.

8.8 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.[11] 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.

[11] Remember the two types of `CMakeLists.txt` files: "top level" and "directory level". The former (see Sec. 8.1) is the potentially big file at the top level of your project. The latter (see Sec. 8.6) is the smaller file in the directory with your actual source code files.

8.8.1 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,[12]

```
...
cet_report_compiler_flags()
...
find_ups_product(art v1_08_10 )
find_ups_product(fhiclcpp v2_17_12)
find_ups_product(messagefacility v1_10_26)
...
```

[12] 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).

8.8.2 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,[13]

```
...
cet_report_compiler_flags()
...
find_ups_root(v5_34_12)
...
```

[13] 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,[14]

```
find_library(ROOT_GLEW NAMES GLEW PATHS ${ROOTSYS}/lib
                    NO_DEFAULT_PATH)
find_library(ROOT_GPAD NAMES Gpad PATHS ${ROOTSYS}/lib
                    NO_DEFAULT_PATH)
find_library(ROOT_GRAF NAMES Graf PATHS ${ROOTSYS}/lib
                    NO_DEFAULT_PATH)
find_library(ROOT_GRAF3D NAMES Graf3d PATHS ${ROOTSYS}/lib
                    NO_DEFAULT_PATH)
```

[14] 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. 8.6.1 for where to put such items in the arguments. For example,[15]

```
        art_make (
           LIB_LIBRARIES
              ${ROOT_GPAD}
           MODULE_LIBRARIES
              ${ROOT_TREE}
              ${ROOT_TVMA}
```

[15] In the example left, regular sources are linked against `libGpad.so` while modules are linked against `libTree.so` and `libTVMA.so`.

```
        )
```

8.8.3 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,

```
...
cet_report_compiler_flags()
...
find_ups_geant4(v4_9_6_p02)
...
```

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. 8.6.1 for where to put such items in the arguments. For example, `srcs/gm2ringsim/calo/CMakeLists.txt` has, in part,[16]

```
art_make(
    LIB_LIBRARIES
        gm2geom_calo
        gm2geom_station
        artg4_material
        artg4_util
        ${XERCESCLIB}
        ${G4_LIB_LIST}
    SERVICE_LIBRARIES
        gm2ringsim_calo
      )
```

[16] If you are curious, you can see where `G4_LIB_LIST` is defined in `$CETBUILDTOOLS_DIR/Modules/FindUpsGeant4.cmake`. `XERCESLIB` goes with Geant.

8.8.4 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/calo` directory, then `CaloHitSD.cc`, when it includes `CaloHitSD.hh`, can do either

```
#include "CaloHitSD.hh"
```

or

```
#include "gm2ringsim/calo/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. 8.6.1 for where to put such items in the arguments. For example, `srcs/gm2ringsim/calo/CMakeLists.txt` has, in part,[17]

```
art_make(
    LIB_LIBRARIES
        gm2geom_calo
        gm2geom_station
        gm2ringsim_station
        artg4_material
        artg4_util
        ${XERCESCLIB}
        ${G4_LIB_LIST}
    SERVICE_LIBRARIES
        gm2ringsim_calo
      )
```

[17] In the left example, services in that directory are linked against the library that gets created from the regular sources, namely `libgm2ringsim_calo.so`. You can predict the name of the library by taking the source directory (e.g. `gm2ringsim/calo`) and replacing the slashes by underscores.

If any source file uses a header that is present in a different directory in your project, then you must link against that library. In the example above, code in the `gm2ringsim/calo` 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.

8.8.5 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.8.1 for an example.

In your directory `CMakeLists.txt` file, you need to list the library corresponding to the code you are using. See Sec. 8.6.1 for where to put such items in the `art_make` arguments. For example, `srcs/gm2ringsim/calo/CMakeLists.txt` has, in part,

```
art_make(
    LIB_LIBRARIES
        gm2geom_calo
        gm2geom_station
        artg4_material
        artg4_util
        ${XERCESCLIB}
        ${G4_LIB_LIST}
    SERVICE_LIBRARIES
        gm2ringsim_calo
      )
```

When the regular sources are built, they will be linked against code in `gm2geom/calo`, `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.

9
Things You May Do in Your Code

This chapter contains some reminders of common things you do in Muon $g-2$ code.

9.1 Dealing with parameters

The constructor for your module or service has the parameter set as an argument. You can retrieve information from the parameter set and supply defaults if the parameter does not exist as in the example below.

```
gm2ex::CalorimeterDigitizer::CalorimeterDigitizer(
            fhicl::ParameterSet const & p) :
  category_    (p.get<std::string>("category","digi")),
  TAURAMP_     (p.get<float>("TAURAMP", 1.4 /* ns */)),
  TAUDECAY_    (p.get<float>("TAUDECAY", 36.4 /* ns */)),
  PULSELENGTH_ (p.get<int>("PULSELENGTH", 30 /* samples */)),
// ...
```

9.2 Readling enviornment variables

```
 #include <cstdlib>
//...
std::string value = std::getenv("PATH'');
```

The argument to `std::getenv` is a constant character array, not a `std::string`.

9.3 Throwing an exception

See http://mu2e.fnal.gov/public/hep/computing/exceptions.shtml.

```
#include "cetlib/exception.h"
// ...
if ( something ) {
```

```
  throw cet::exception(CATEGORY) << "Message\n"
}
```

9.4 Finding a file

`cetlib` has a nice facility for searching for files in a path specification. See `$CETLIB_INC/cetlib/search_path.h`.

It may be convenient to specify the search path in a FHICL parameter with the possibility of providing an environment variable. Here is some code that takes a search path through the parameter, but if the first character is a $, it then gets the path through the specified environment variable.

```
gm2util::MetadataFromFile::MetadataFromFile(
          fhicl::ParameterSet const & p) :
    searchPath_  (p.get<std::string>("searchPath", ".")),
    fileName_    (p.get<std::string>("fileName")),
    keyName_     (p.get<std::string>("keyName"))
{
  // Let's parse the search path
  // If the first character is a dollar sign, then the
  // remaining is an environment variable
  if ( searchPath_.at(0) == "$" ) {
    std::string envVar = searchPath_.substr(1);
    char* envValue = std::getenv(envVar.c_str());
    if ( ! envValue ) {
      searchPath_ = ".";
      throw cet::exception("META_DATA_FROM_FILE") <<
       "Environment␣variable␣" << envVar << "␣is␣not␣set";
    }

    searchPath_ = std::string(envValue);
  }
}
```

10
Frequently Asked Questions

Some questions are answered here that didn't seem to fit in other sections.

Where is the art source code? The art source code[1] for a particular `gm2` release is accessible in our release area for you to peruse. Set up the the release (see section 3) and look in `$ART_DIR/source/art`.

[1] Never use the source code directory for an `#include` in your code. Instead, just use `#include "art/whatever.h"` and the build system will find it in `$ART_INC`.

Index

add_subdirectory, 43
art_make, 44
  arguments, 45
artmod, 43

CMakeLists.txt
  directory level, 44
  top level, 41

exceptions, 53
external code, 46

find_ups_geant4, 49
find_ups_product, 47
find_ups_root, 48

input source
  writing, 44

install_headers, 44

linking, 46

modules
  writing, 43

services
  writing, 44