

MUON *g* – 2

OFFLINE COMPUTING
AND SOFTWARE
MANUAL
[GM2 V6_03_00]

July 22, 2016

Git version: v6_03_00-0-g74be1a0

[GM2-doc-1825](#)

Quick Start Guide

Send suggestions to jstaplet@fnal.gov

Setup g-2 software (Section 3.1)

```
. /cvmfs/gm2.opensciencegrid.org/prod/g-2/setup   Prepare shell for software from CVMFS
setup gm2 v6_03_01 -q prof                       You now have the gm2 and mrb commands
```

Create/Setup a development environment (Section 3.2)

```
mkdir DEVAREA; cd DEVAREA; mrb newDev           Create new local development area
. DEVAREA/localProducts_gm2_v6_03_00_prof/setup Prepare shell to use local development area
```

Get/Build code using mrb (Section 3.3)

```
cd DEVAREA/srcs; mrb gitCheckout PACKAGE       Checkout PACKAGE into srcs/ directory
. mrb setEnv                                   Prepare shell to compile checked out PACKAGE(s)
mrb build                                       (Re)Build everything in local development area
```

Contents

1	<i>Introduction</i>	7
1.1	<i>What code goes with this document?</i>	7
1.2	<i>The Release Philosophy</i>	8
2	<i>Setting up a Development Environment</i>	11
2.1	<i>Your options</i>	11
2.2	<i>Fermilab gm2 group Virtual Machines</i>	12
2.3	<i>Native development on your own computer</i>	13
2.4	<i>CentOS Virtual Machine</i>	14
2.5	<i>Docker container</i>	14
3	<i>Developer Workflow</i>	15
3.1	<i>Setup/Initialize gm2 software environment</i>	15
3.2	<i>Setting up a development area</i>	15
3.3	<i>Checking out and building code</i>	16
3.4	<i>Incremental rebuilds</i>	17

4	<i>Using a Mac for Development</i>	21
5	<i>Running the simulation</i>	23
5.1	<i>Component packages in the simulation</i>	23
5.2	<i>Using a base release</i>	23
5.3	<i>Using a point release</i>	23
5.4	<i>FCL files for the simulation</i>	24
6	<i>Running Jobs on the Grid</i>	25
6.1	<i>First Learn about jobsub</i>	25
6.2	<i>Output location</i>	25
6.3	<i>Generating Data</i>	26
6.4	<i>gridSetupAndSubmit.sh</i>	26
6.5	<i>submit-release.sh</i>	27
6.6	<i>submit-localrelease.sh</i>	27
7	<i>Storing Data and Using SAM</i>	29
7.1	<i>What is SAM?</i>	29
7.2	<i>File Locations</i>	29
7.3	<i>User Datasets</i>	30
7.4	<i>Running over a Dataset</i>	31
7.5	<i>Automatic SAM Database population</i>	31

8	<i>Writing Source Code</i>	33	
8.1	<i>Top level CMakeLists.txt file</i>	33	
8.2	<i>Organizing Source Code</i>	35	
8.3	<i>Writing Modules</i>	35	
8.4	<i>Writing Services</i>	36	
8.5	<i>Writing Input Source Modules</i>	36	
8.6	<i>Directory level CMakeLists.txt file</i>	36	
8.7	<i>Libraries produced from building</i>	38	
8.8	<i>Using External Code (Linking)</i>	38	
9	<i>Things You May Do in Your Code</i>	45	
9.1	<i>Dealing with parameters</i>	45	
9.2	<i>Reading environment variables</i>	45	
9.3	<i>Throwing an exception</i>	45	
9.4	<i>Finding a file</i>	46	
10	<i>Frequently Asked Questions</i>	47	
11	<i>Releases of gm2</i>	49	
11.1	<i>gm2 v6_01_00 -q prof and (-q debug)</i>	49	
11.2	<i>gm2 v6_00_00 -q prof and (-q debug)</i>	49	
11.3	<i>gm2 v5_01_00 -q e6:prof</i>	51	
11.4	<i>gm2 v5_00_00 -q e6:prof and (-q e6:debug)</i>	51	
11.5	<i>gm2 v201402 -q e4:prof</i>	52	
11.6	<i>The Release Philosophy</i>	52	

12	<i>What is this document?</i>	55
12.1	<i>What code goes with this document?</i>	55
12.2	<i>Obtaining this documentation</i>	56
12.3	<i>Obtaining the source for this documentation, contributing to it, and building it</i>	56
	<i>Index</i>	61

1

Introduction

This document is meant to be a user's manual to the Muon g-2 offline and simulation software and computing system. It 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. This PDF file 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.

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 latest official version of this documentation is in GM2 DocDB as [GM2-DOC-1825](#).¹ Newer releases of `gm2` (starting from `gm2 v5_01_00`) will have a copy of this manual that corresponds to the particular `gm2` version at `$(GM2SWDOCS_DIR)/manual.pdf`.

¹ DocDB uses its own versioning scheme (just a sequential number) which does not correspond to the `gm2` release.

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 v6_01_00`. On the bottom of the title page is the git version information for this document itself. For this version, it reads `v6_01_00_00-9-g556979a`. 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 The Release Philosophy

What is a `gm2` release? A `gm2` release is a versioned collection of libraries and executables that you either use or build your code against. A particular `gm2` release contains a particular version of `gcc`, `art`, `root`, `geant4`, etc. These libraries/executables are called the *externals*. A `gm2` release may also contain `g-2` applications and libraries built against those externals (e.g. `gm2ringsim`, `artg4`). If the versions of those packages are suitable for you, then you can use them directly without having to build them yourself. This means we have *official* versions of these packages.

Official releases are important. For the purpose of scientific reproducibility, it is important to know how results were produced. Using a versioned release means that we know the code used for an analysis and can re-run it to do further analyses or look for mistakes. Official releases are essential for sharing code, as it gives people a common base and starting place.

The philosophy of `gm2` releases is that the first (major) version number in the release (e.g. the 5 in `v5_00_00`) is the release *series*. Releases in the same series are built with the same version of externals (`gcc`, `art`, `geant4`, `root`, etc) and so they are all binary compatible. The `vX_00_00` release only has externals in it and is called a *base* release. We then add point releases (e.g. `v5_01_00`, `v5_02_03`) containing `g-2` libraries and applications (e.g. `gm2ringsim`). If there is a new `art` or `root`, then the major version number increases (e.g. to `v6_00_00`) and a new series is started. New major releases (new series) should occur only 3-4 times per year.

The point releases can occur more often and represent official

changes to the $g - 2$ code base (e.g. new geometry or features in the simulation). Feature changes advance the middle (minor) version number and bug fixes advance the last (patch) version number. So the first release of $g - 2$ code for a new series is `vX_01_00`. A feature addition will advance to `vX_02_00`. A subsequent bug fix will advance to `vX_02_01`.

Users adoption of these point releases is optional. They can always build all of the necessary code based on the `vX_00_00` base release. But using a point release can be convenient and save a large amount of time by using pre-built libraries instead of building them by hand. The point releases also represent a trackable official progression of features and bug fixes. Users can also use libraries from a point release, but build parts of the release themselves that they are developing (e.g. developing `gm2ringsim`, which is also in the release). In these cases, the build system will automatically use the user developed code instead of what is in the release. The `superbuild` system, which does builds across platforms for use on the grid, will automatically mark such user-built libraries as unofficial.

1.2.1 How do releases help me as a user/developer?

If the official released code is suitable for you (e.g. you are not developing the code in the release, but are instead developing code that *uses* the release), then using an official release will save you compilation time and will be more convenient. You can more easily track what you have run on the grid. You may be able to run without having to build anything (e.g. simply running the official `gm2ringsim`).

You should use an official point release whenever possible. Instructions for setting up your development area and how to migrate to new releases are given in the section under the release notes as well as in Chapter 3 of this manual.

2

Setting up a Development Environment

A $g - 2$ development environment consists of an operating system, a suite of software tools for compilation and package management, the $g - 2$ software itself, and the software packages upon which it depends. All of the software is distributed over the CVMFS network file system.

Our software is built upon the `art` event-processing framework, which is only supported on specific versions of Linux and Mac OS X. If you are not using one of these operating systems then you will need to either work remotely by logging in to pre-configured systems or set up operating system virtualization on your own computer.

2.1 Your options

There are at least four different methods which you can use to setup a development environment with access to the $g - 2$ software:

- $g - 2$ group Virtual Machines (`gm2gpvmNN.fnal.gov`)
- native development on your own computer (only certain versions of Mac/Linux)
- running a CentOS Virtual Machine (VM) on your own computer
- running a Docker container on your own computer

Their relative strengths and weaknesses will determine the best choice for you. The $g - 2$ group Virtual Machines (VMs) only require Kerberos and SSH (or Putty) for logging in to preconfigured systems which reside onsite at Fermilab. This can be the easiest option to set up, but requires a good/fast connection to `fnal.gov` and provides relatively poor computing performance. Your own laptop or desktop may run the software faster, but this is only supported for certain versions of Mac OS X or Linux and also requires installation of a special network file system. If you can install VirtualBox and Vagrant on your computer then it can be relatively simple to use a CentOS Virtual Machine provided by the $g - 2$ collaboration. A lighter-weight solution may be provided by the Docker toolkit, which is a new (and experimental)

approach which lies somewhere between native development and a Virtual Machine. The next sections will cover these in more detail.

All four development environments discussed in this chapter will provide access to the $g - 2$ software releases via a network filesystem called the CernVM File System (CVMFS), which is used by many HEP experiments to distribute software to collaborators. It is optimized for the distribution of files and metadata from web servers directly to a virtual file system mounted at `/cvmfs` in the end-user's development environment.¹ Its focus on read-only software distribution allows it to avoid firewall problems typical to other network file systems and to aggressively cache directory contents in a way that is nearly invisible to the user. You can read more about it at <https://cernvm.cern.ch/portal/filesystem>.²

Follow the instructions in one of the following sections to set up your development environment, and then go to Chapter 3.

2.2 Fermilab $gm2$ group Virtual Machines

These Virtual Machines are already set up on computers on the Fermilab network.³ If you have not yet set up Kerberos for network access at Fermilab, you can find instructions by going here: <https://cdcvs.fnal.gov/redmine/projects/gm2cdr/wiki/InstallingKerberos>. You must install Kerberos and download/install a configuration file which tells it about the FNAL.GOV domain. Fermilab uses this system for site-wide network authentication. You must use your Fermilab username and password to authenticate by typing `kinit your_fnal_username@FNAL.GOV` (case sensitive) into a terminal on your own computer and entering your password when prompted.⁴ Kerberos tickets expire after a preset time (currently 24 hours), after which you will have to run the `kinit` command again and enter your password.

Nearly every Mac or Linux computer will have OpenSSH installed, which provides the `ssh` command which we use to open a terminal session on a remote computer (or 'host'). Windows users can install Putty, which provides much of the same utility, but some of the following instructions may need to be adapted.

There are certain configuration options needed for SSH which tell it how to authenticate with the $g - 2$ group virtual machines. These options are placed in a text file (`.ssh/config` in your home directory on Mac/Linux):

```
Host gm2gpvm*
  ForwardX11 yes
  ForwardX11Trusted yes
  GSSAPIAuthentication yes
```

¹ The CVMFS caching strategy has the effect of hiding directories which have not yet been accessed within the user's virtual file system. A consequence of this is that checking the top-level directory with `ls /cvmfs` will show only subdirectories which have recently been accessed. If you check `/cvmfs` and do not see the subdirectory `gm2.opensciencegrid.org`, then try `ls /cvmfs/gm2.opensciencegrid.org` before you conclude that it is not there.

² Note that we currently use the subdirectory `/cvmfs/gm2.opensciencegrid.org/`. The subdirectories `oasis.opensciencegrid.org` and `fermigrid.opensciencegrid.org` are deprecated. Check your paths, and be sure not to use them!

³ This is different from running the Virtual Machines on your own computer, which is covered in later subsections.

⁴ This obtains a Kerberos ticket, stores it on your computer, and automatically uses it when you attempt remote login to *any* Fermilab computer.

```
GSSAPIDelegateCredentials yes
GSSAPITrusDns yes
#PreferredAuthentication gssapi,gssapi-with-mic
```

Add this to the config file if it already exists. The last option is necessary in some cases but not others, depending on your version of OpenSSH. If your login is rejected then uncomment it by removing the *pound* symbol (#). You can then login to `gm2gpvm02`, for example, by typing

```
ssh your_fnal_username@gm2gpvm02.fnal.gov
```

At your first login you will be prompted to accept an RSA key (respond with `yes`).

You can find more information at <https://cdcvs.fnal.gov/redmine/projects/g-2/wiki/ConfiguringSsh> and <https://cdcvs.fnal.gov/redmine/projects/g-2/wiki/GPCF>, though these may be somewhat outdated.

2.3 Native development on your own computer

The `art` event-processing framework will run natively on Scientific Linux Fermi (SLF) versions 5, 6, and 7, and Mac OS X up to (but **not** including 10.11).⁵ The `g-2` software is tested on SLF6 and OS X, so we recommend one of these. Other Linux distributions known to work are Scientific Linux (*without* the ‘Fermi’), Red Hat Enterprise Linux, and CentOS. Modern versions of Fedora Linux and SuSE Linux may also work.⁶ Debian-based variants such as Ubuntu are not supported (but may be in the near future).

If you are using one of these operating systems, then you only need to install the tools required to use the CVMFS network file system. The required software is provided by CERN at <http://cernvm.cern.ch/portal/filesystem/downloads>. Installation instructions are provided below.

⁵ Note that OS X 10.11 (El Capitan) is **NOT** supported due to System Integrity Protection (SIP), Apple’s new security ‘feature’.

⁶ The Linux distribution ecosystem produces many different types of Linux-based operating systems which may have major or minor similarities or differences, and these differences range from fundamental system structure to superficial packaging choices. The Red Hat distribution serves as a starting point upon which many other distributions are built.

2.3.1 CVMFS installation on Linux

Instructions can be found here: <https://twiki.grid.iu.edu/bin/view/Documentation/Release3/InstallCvmfs>.

2.3.2 CVMFS installation on Mac OS X

Instructions can be found here: <https://cdcvs.fnal.gov/redmine/projects/g-2/wiki/InstallingOasisOnMacLaptop>.

2.4 CentOS Virtual Machine

You can set up a Virtual Machine on your computer which includes a supported operating system, access to CVMFS, and all of the required software tools. After it is set up, it will compile and run the $g - 2$ software using your own CPU.

First you need to install [VirtualBox](#) and a Virtual Machine setup tool called [Vagrant](#). Then go to Adam Lyon's [centos-gm2-dev Github page](#), where you can find complete instructions for using Vagrant to set up a fresh CentOS VM. Be sure to follow the command-line instructions for installing the `vagrant-guest` plugin, clone the github repository, and provision the VM. This can take some time, as Vagrant starts with a small CentOS system image, updates it, and runs some additional scripts.

After this, you can start the Vagrant VM by typing the command `vagrant up`. This must always be done in the same directory as the file `Vagrantfile` which lives in the `centos-gm2-dev` directory. You can log in to the virtual machine by simply typing `vagrant ssh`.⁷ Multiple shells may be opened simultaneously simply by executing `vagrant ssh` in another terminal. You can pause the virtual machine with `vagrant suspend`, and shut it down completely with `vagrant halt`. You can check the status of the VM with `vagrant status` (as well as `vagrant global-status`).⁸

The Vagrant command-line utility has very complete help output, which you can access by simply typing `vagrant`. Help for a particular command can be accessed by `vagrant COMMAND -h`.

2.5 Docker container

[TODO]

⁷ Note that you will have to run the command `vagrant up` each time the VM has been stopped.

⁸ Many of these commands accept an ID for the VM; this is the hexadecimal code listed in the first column of output from `vagrant status`.

3

Developer Workflow

After you have finished setting up your development environment (see Chapter 2), you will have access to the $g - 2$ software at `/cvmfs/gm2.opensciencegrid.org`.

3.1 Setup/Initialize *gm2* software environment

Assuming the CERN Virtual Machine File System is mounted at `/cvmfs`¹, you can get started with

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

This will print some information about release versions of the `gm2` software, including the appropriate command which you should use now to set up the latest release:

```
setup gm2 v6_03_00 -q prof
```

At this point you can invoke `art` through the command `gm2`, which accepts the same command-line arguments as the `art` executable. For example, the Mock Data Challenge Zero simulation can be run by executing `gm2 -c mdc0.fc1` and all output files will be placed in your current working directory. This is adequate for basic use of the $g - 2$ analysis tools *using a stable release version*. Continue to the next section if you will be modifying the source code for `gm2` or one of its dependencies (`gm2ringsim`, `gm2dataproducts`, `artg4`, etc).

NOTE: The commands above must be run for **every** new login session because they modify the shell environment.

3.2 Setting up a development area

The instructions above set up pre-compiled versions of the $g - 2$ software packages which are accessible from `/cvmfs`. However, active development of this software requires a local build environment for

¹This will be true on the $g - 2$ group virtual machines (`gm2gpvm`), and for some other specialized environments. Be sure to use the latest CVMFS directory (`/cvmfs/gm2.opensciencegrid.org`) instead of the deprecated locations containing `oasis` or `fermiapp`.

testing changes to the code.² The `ups` packaging system and `mrB` build scripts allow you to check out and build code for a particular package, relying on pre-compiled code in `/cvmfs` for packages which are not under development. This allows developers to work on only part of the $g - 2$ analysis code (e.g. `gm2ringsim`) without having to build everything else.

After the environment setup from the previous section, we use `mrB` to setup and initialize a development directory and shell build environment. The development directory can have any name or be at any location you choose.³ A new development area `DEVAREA` can be setup quickly by doing⁴

```
mkdir DEVAREA
cd DEVAREA
mrB newDev
```

This sets up source, build, and local product directories. You will typically only work in `DEVAREA/srcs/`, while `mrB` will use the other directories in the background. The only exception to this is a setup script in the local products area which **must be sourced now**:

```
source localProducts_gm2_vXX/setup
```

This sets up the environment to prefer local versions of some packages (instead of relying on the precompiled version in `/cvmfs`). NOTE: This step must be run for **every** new login session as it modifies the shell environment.

3.3 *Checking out and building code*

The development area is now set up, though it contains no source code to build. To download the source code for a software package and set it up to work within the $g - 2$ ecosystem, move to `DEVAREA/srcs/` and use `mrB` to check out code for that package:

```
cd srcs
mrB gitCheckout PACKAGE
```

where `PACKAGE` is replaced with the desired package name (e.g. `artg4`, `gm2ringsim`, etc.). This creates a subdirectory named `DEVAREA/srcs/PACKAGE`. If you need to check out multiple packages, run `mrB gitCheckout PACKAGE` for each, always from the `DEVAREA/srcs/` directory.⁵

Now that the source code is in place, we finish setting up the build environment for `mrB`:

```
source mrB setEnv
```

² NOTE: Rebuilding the software is **NOT** necessary if your analysis only requires changes to FHiCL files (changing module parameters, analysis paths, or re-running analysis using different input files or numbers of event). Rebuilding is **required** if you must change source code for an `art` module or service.

³ This location may require large amounts of storage space. If on `gm2gpvm` please keep application development in `/gm2/app/users/yourusername` and manual storage of large files in `/gm2/data/users/yourusername`. This automatically stores files elsewhere using the Network File System (NFS). (NOTE: If the subdirectories `yourusername` do not already exist in these locations, you can probably create them using `mkdir`.)

⁴ If the `mrB` command is not available, then you probably need to set up the `gm2` software environment using the instructions from the previous section.

⁵ If you must use `git flow` to check out a feature branch for a particular package, you have to `cd PACKAGE` in order to run the `git` command *in the* `DEVAREA/srcs/PACKAGE` subdirectory.

This parses all of the `product_deps` files for your packages and determines dependencies. It then sets them all up from ups (e.g. `geant4`). You must run this command every time you start or resume a development session (e.g. log out and back in again later). You should not see any errors.

3.3.1 Building for the first time

When you have no build products at all in your build directory (`$MRB_BUILDDIR`), start the build with

```
mr b
```

That will run `cmake`⁶ and then `make`⁷ to build your code. The build will stop if there are any `cmake`, compilation, or linker errors.

⁶ See <http://www.cmake.org/> for more information.

⁷ `make` is the standard build tool that determines dependencies, build order, and issues the commands. `make` uses *Makefiles* for configuration and construction. These *Makefiles* are extremely difficult to write correctly. `cmake` is a tool with a simpler configuration language that will write all of the *Makefile*'s for us.

3.3.2 Running tests

If packages you are building contain tests, you can run them with,

```
pushd $MRB_BUILDDIR # cd to that directory and push to stack
ctest -j N          # N is number of CPU cores
popd                # Change back to old directory
```

Where `N` is equal to or less than the number of cores on your machine (use 1 for `gm2gpvm`; a newer Mac laptop may have as many as 8 cores). The `-j` option is optional, but if you give it the tester can run tests in parallel and will be much faster. Your current directory must be `$MRB_BUILDDIR`.

If a test fails, look in `$MRB_BUILDDIR/Testing/Temporary` for log files.

3.3.3 Re-building incrementally

When you make a change to your code, you need to build it again (an incremental build). The build system can figure out what has changed and only rebuild the modified code and anything that depends on it. You can do this by re-running `mr b`. Note that this will re-run `cmake` perhaps unnecessarily. See below for faster ways to rebuild.

3.4 Incremental rebuilds

If you have not changed any `CMakeLists.txt` files and you have not added any new header files, you can skip the `cmake` step on an incremental re-build by doing,

```
pushd $MRB_BUILDDIR # cd to that directory and push to stack
```

```
make -j N          # N is number of CPU cores
popd              # Change back to old directory
```

This may still take a minute or two as `make` has to check each directory for changes (see below for faster methods). Use `-j` to specify the number of cores on your machine to do builds in parallel. On `gm2gpvm`, leave off the `-j` since there is only one core.

3.4.1 Incremental rebuild (*super-fast but potentially dangerous*)

When `make` runs, it tells you the *target* that is building. If you know the name of the target for your build, you can tell `make` to only make that target. For example,

```
pushd $MRB_BUILDDIR # cd to that directory and push to stack
make gm2artexamples_Lesson2_makeRotatedHits_module
popd                # Change back to old directory
```

This technique is somewhat dangerous, as `make` will not build other targets that depend on the one you have changed, possibly leading to an inconsistent and incorrect build. But, if you are changing an art module which cannot have downstream dependencies, then you are safe in only building that module's target.

This partial build is very fast, as you are telling `make` to only build a very small part of the codebase.

3.4.2 Building with *ninja* (*amazingly fast and apparently safe*)

`ninja`⁸ is a build system that replaces `make`. Fortunately, just as `cmake` knows how to create the files necessary for `make`, `cmake` also knows how to create the files for building with `ninja`. `ninja` works on all platforms.

⁸ See <http://martine.github.io/ninja/>

The advantage of `ninja` over `make` is that if you do an incremental build, `ninja` can determine what files need compiling in practically zero time.

For example, if you do a full build, and then do an incremental build (with `mr b`) changing nothing, the build will take quite awhile to figure out that there is nothing to do. That is because `make` needs to check each directory for updated files. Somehow, `ninja` figures this out a different way.⁹

Compiling and linking takes time, of course, but you will get there much faster.

Though I have done builds with `make` and `ninja` and see no problems with using `ninja`, currently `ninja` is experimental and you will have to follow some extra steps to use it.

⁹ I think `ninja` polls the file system event logger to determine what files were updated and it will return instantaneously.

First build with ninja

`ninja` replaces `make`. You can decide to use `ninja` for your build directory. If you've already done a build with `make`, you must zap it (delete it with `mrbs; . mrbs`) and then redo the full build with `ninja`. Once you've built with `ninja`, you must zap again to go back to `make` for that build directory. The upshot is that you cannot freely switch between `make` and `ninja` for a build directory.

Because `ninja` is experimental, you must set it up explicitly. Before running the build, do (you will need to do this each time you log in),

```
setup ninja v1_5_3a
```

Now, you need to do a full build with,

```
. mrbs
mrbs --generator ninja
```

You won't see much speed up here, as this is a full build. The speed up occurs for incremental builds.

Incremental builds with ninja

You must have done a full build with `ninja` as described in the previous section. If you have logged out and logged back in in the meantime, re-run the `setup ninja` command above.

Now when you change some code and want to do an incremental build, do

```
pushd $MRB_BUILDDIR # cd to that directory and push to stack
ninja              # The magic happens
popd               # Change back to old directory
```

`ninja` will figure out the number of cores you have. `ninja` will determine all of the files that need to be re-compiled and linked with almost no overhead.

4

Using a Mac for Development

Information and instructions for developing and running $g - 2$ code on the Mac has moved to its own document. See [GM2-doc-2459](#).

5

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 ??.

5.1 Component packages in the simulation

Our simulation code is made up of four packages:

- `artg4` - serves as the interface between the `art` framework and `geant4`.
- `gm2geom` - a prototype geometry server
- `gm2dataproductions` - data products used for emitted by the simulation
- `gm2ringsim` - the simulation code itself

5.2 Using a base release

See section 11.6 for the meaning of point and base releases. The base release (e.g `v5_00_00`) only has libraries and executables for the external programs. Therefore to run the simulation from a base release, you must build all of the component packages yourself.

5.3 Using a point release

See section 11.6 for the meaning of point and base releases. With a point release, you may use some or all of the component packages out of the release instead of building them yourself. You should check the `CHANGELOG` (e.g. `less $GM2RINGSIM_DIR/CHANGELOG`) to make sure that the packages were built with the features you want. If so, then simply run a FCL file with `gm2`; no need to build anything or even set up a development area.

If you need to build a package because you want to run something even newer than what was released or you have changes, then follow the dependency tree. In section 5.1, we see the list of components. This was purposefully written to show dependencies from bottom up. E.g. `gm2ringsim` is at the bottom. If you only need to change `gm2ringsim` then you only need to checkout and build your version of `gm2ringsim`. `gm2ringsim` depends on `gm2dataproducs`, so if you change something in `gm2dataproducs` you will need to checkout and build `gm2dataproducs` and `gm2ringsim`. So the way to read that list of components in section 5.1 is that if you change and build a package, you must also change and build everything below it on the list.

5.4 FCL files for the simulation

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.

6

Running Jobs on the Grid

In this section you will learn how to run your jobs on the grid at Fermilab. You can either run the latest code that is released, or you can run from a local release.

For official simulation runs you should make sure you are running the code out of a release. If you are doing some tests and need a certain configuration, for example you put trackers in all 24 slots because you want specifically to see information about tracking and could use more tracker events, then you would run out of a local release.

6.1 First Learn about jobsub

Before going in to the details on what we will do specifically for $g - 2$ it would be good to understand a little bit about jobsub. There are lots of details here: <https://cdcvs.fnal.gov/redmine/projects/jobsub/wiki> and this PDF is a good overview:

https://fermipoint.fnal.gov/project/FIFE/Shared%20Documents/FIFE_Jobsub_tutorial.pdf.

6.2 Output location

Before starting with the grid you'll need a place for the output to go. You will use an area on `/pnfs/`. You will put your stuff in `/pnfs/GM2/scratch/users/your_fnal_user_name` (Using your Fermilab username will help if you choose to use scripts provided to run your jobs).

After testing your output and verifying it, it will be moved over to a taped back location. *[This information will be added when we have some official data/simulation we want to save]*. See the following section for information on how to analyze your output using SAM.

Let's get to creating your data!

6.3 *Generating Data*

This is very simple!

Log into your favorite gm2 machine and do the following:

```
source /cvmfs/gm2cfs.fnal.gov/prod/g-2/setup
```

Followed by the version you want to run. For example:

```
setup gm2 v6_01_00 -q prof
```

Go to your `/gm2/app/users/username` area and create an area to run your jobs from.

Now you want to set things up so that you can run on the grid:

```
source /grid/fermiapp/products/common/etc/setups.sh
setup jobsub_client
```

There are three scripts that you can use for submissions located here: `/gm2/app/users/leah/submitGridJobs/`. They will be in git soon.

6.4 *gridSetupAndSubmit.sh*

This file sets things up for the producing. Following are descriptions of the variables that are set in this file:

- `SCRATCH_DIR`: sets things up so your output will go to the pnfs area: `/pnfs/GM2/scratch/users/username/NOW`
- since `$USER` is a environment variable you can just use it in the script which is why it was suggested you name the area on pnfs with your user name.
- If you want a different directory structure that's to your discretion.
- `MAINFCLNAME`: whatever fcl file you want to run.
- `NEVTSPERJOB`: how many events in each submission you plan on running
- `NJOBS`: how many of such jobs
- [Each are set to be 1 as a way to test things before submitting thousands of events/jobs]*

To actually submit the job, run the `jobsub` command and send in either the `submit-release.sh` or `submit-localrelease.sh`. See below for the differences between these files.

You give it the following options:

- `-N`: Number of jobs
- `-G`: gm2 (for our group name)
- `-M`: Asking to send me an email when the job finishes

- `--OS`: SL6 (use only Scientific Linux 6 machines)
- `--resource-provides`:
- `--role`: Analysis
- This is all you have permission for right now, eventually Production will be used as well.
- `file`: This is the actual script that will submit to the grid. Note that after the file are more variables that have been previously set. These are variables that the script is going to use. If your script doesn't have such things they aren't necessary.

6.5 *submit-release.sh*

The only thing in this file that you have to change is the release you want to use (ex: `setup gm2 v6_01_00 -q prof`):

6.6 *submit-localrelease.sh*

There are just a few extra steps if you want to run some jobs based on some code you have in a local area.

IMPORTANT: First you have to install the local products. This is simple just: `mrB i`

Once you do this, then when you run the job on the grid you'll have to setup your local products and that can't be done if you don't first install them. The only difference from running from an official release and a local release setting up your local area instead of the official release:

```
ex: source /gm2/app/users/leah/gm2Dev_v6_01_00/localProducts_gm2_v6_01_00_prof/setup
Followed by setting up the local products: . mrB slp
```


7

Storing Data and Using SAM

In this section you will find information on how to add your data to the SAM (Sequential Access via Metadata) database.

7.1 What is SAM?

Simply speaking, SAM is a data handling system. As physicists we are going to produce a large amount of data and simulation that needs to be organized. Keeping this data in folders on hard drives is not a scalable option. And even if it was, it's hard to find data once it's in those nested folders. SAM indexes all the data it has by meta-data and is agnostic to the location of the files. Making it easier for you, the user.

So generally SAM does it all:

- Keeps track of the locations of files on tape and disk.
- Handles file meta-data so users do not need to know the file name to find data of interest
- Delivers files without users having to know where they come from
- Bookkeeping: keeps track of datasets created and files processed while analyzing output.

So, the first thing we are going to have to do is make some datasets that we can use.

7.2 File Locations

Before you get started, you'll need a place to store your data. We will store all the data in the `/pnfs/` system. To start with we are going to use the scratch area: `/pnfs/GM2/scratch/users/yourUserName`. Eventually if you don't touch files here for a few months they will disappear. If you are continually using them then there is no problem. If you know your files are good and something you want forever, they will go to a taped back location. This is simply any area under

/pnfs/GM2/ that isn't under scratch. **More information coming on this, please don't (at this point) put anything on tape.**

7.3 *User Datasets*

A user defined dataset is just what it sounds like it. It's a set of data which has common attributes (meta-data) defined by the user.

You will be able to refer to this dataset, when you want to run an analyzer, by name instead of worrying about where all the files are. Because really you don't care where they are, you just care about what they contain in them.

NoVA has some good documentation available that is summarized below: https://cdcvs.fnal.gov/redmine/projects/nova_sam/wiki/User_Datasets

7.3.1 *Setup*

Before you start doing anything you'll need the FIFE Utilities package. Set this up by doing:

```
>setup fife_utils v2_x [The NoVA link is updated for the latest
version of the fife_utils]
```

You also have to make sure you have a X509 certificate. Get one of these by doing the following two commands

```
>kinit
>kx509
```

You'll need an environment variables set:

```
>export SAM_EXPERIMENT=gm2
```

7.3.2 *sam commands and sam_web*

Now you should have access to SAM commands:

- `sam_add_dataset`: makes a new dataset
- `sam_retire_dataset`: retires a dataset
- `sam_validate_dataset`: validates that all the files are present, or which aren't
- `sam_clone_dataset`: Makes a replica of a dataset in a different location.
- `sam_unclone_dataset`: Removes the replicas of a dataset in a specific location
- `sam_modify_dataset_metadata`: Applies or modifies the metadata associated with a dataset

As well as samweb. Find what's available in samweb by typing:

```
samweb --help-commands
```

7.3.3 *Creating a Dataset*

Finally what we want to do. Creating a dataset out of our data that we've produced.

Something you'll want to take into consideration when creating your dataset. Name it something that is human readable. For example: `lwelty_muplus_062415_gm2v60100`. **Not** things like: `my_data`, `my_good_data`, `use_this_dataset`.

Onward, create that dataset. From a directory where your files are:

```
sam_add_dataset -d <path to file> -n <name of dataset>
```

For example:

```
sam_add_dataset -d . -n leahtestdata
```

Great! That's it. You will notice that if you do an `ls` on your files they will have changed names. By creating a data definition, the names of the files have a prepended hash on the original file name. So for example:

```
gm2ringsim_ProductionMuPlusMuonGasGun_2207507.6.root
```

becomes

```
3610444f-f452-4713-a191-907863dd9cc3-gm2ringsim_ProductionMuPlusMuonGasGun_2207507.6.root
```

This is done because files in SAM cannot have the same name. So this guarantees uniqueness of file names. And we don't really care what the file names are because now you can refer to them simply by your dataset definition name (in my example case, `leahtestdata`).

7.4 *Running over a Dataset*

There are scripts in the `gm2analyses/ProductionScripts/analyze` area that will allow you to run on your SAM datasets. To run simply:

- `./analyzeSAMDataset.sh SAM_datasetname [analyzer(fcl filename)] [fermilab_username]`
- It will run a default of the `AllHits.fcl` file as the analyzer, but you can either give the script the analyzer you want to run on the command line, or just change it in the script itself.
- The output files will show up in `gm2analyses/output/`

7.5 *Automatic SAM Database population*

coming soon.

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 directory¹ 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 `mrbs newProduct` command will create a skeleton file for you.

³ There are five main parts of the file (roughly in order in the file)...

```
1 # Ensure we are using a modern version of CMake
2 CMAKE_MINIMUM_REQUIRED (VERSION 2.8)

4 # Project name - use all lowercase
5 PROJECT (gm2analyses)

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} )

16 # art contains cmake modules that we use
17 set( ART_VERSION $ENV{ART_VERSION} )
18 if( NOT ART_VERSION )
19     message( FATAL_ERROR
20         "ERROR: setup_art_to_get_the_cmake_modules" )
```

- 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

```

21 endif()
22 set( CMAKE_MODULE_PATH $ENV{ART_DIR}/Modules
23      ${CMAKE_MODULE_PATH} )

25 # Import the necessary macros
26 include(CetCMakeEnv)
27 include(BuildPlugins)
28 include(ArtMake)
29 include(FindUpsGeant4)

31 # Configure the cmake environment
32 cet_cmake_env()

34 # Set compiler flags
35 cet_set_compiler_flags( DIAGS VIGILANT WERROR
36     EXTRA_FLAGS -pedantic
37     EXTRA_CXX_FLAGS -std=c++11
38 )

40 cet_report_compiler_flags()

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)

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)

52 # This project uses code from gm2ringsim,
53 # gm2dataproducs, and gm2geom
54 find_ups_product(gm2ringsim v1_00_00)
55 find_ups_product(gm2dataproducs v1_00_00)
56 find_ups_product(gm2geom v1_00_00)

58 # This project uses code from Root
59 find_ups_root(v5_34_12)

61 # Make sure we have gcc
62 cet_check_gcc()

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

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)

```

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 Book⁶ for more information. Only reminders will be given here.

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,

⁶

```

1   artmod producer tracking:TrackFinder
2   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

```

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. 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 `mrbs install`. To do that, you need a line in the directory `CMakeLists.txt` file with

```

1   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),

```

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 )

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}

```

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.

```
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.

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

```
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     )
```

- 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,⁹

```

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

```

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/calor`. Regular sources (not modules, services, nor input sources) get compiled and the objects go into a library called `libgm2analyses_calor.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_Calor_module.cc` then that code will go into a library called `libgm2analyses_calor_Analyze_Calor_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

```

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 ...

```

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

```

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

```

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

```

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)

```

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

```

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

```

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

```
7         )
```

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,

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

```
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    )
```

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/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. 8.6.1 for where to put such items in the arguments. For example, `srcs/gm2ringsim/calor/CMakeLists.txt` has, in part,¹⁷

```

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     )

```

¹⁷ 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.

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/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.

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/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.

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.

```
1 gm2ex::CalorimeterDigitizer::CalorimeterDigitizer(  
2     fhicl::ParameterSet const & p) :  
3     category_    (p.get<std::string>("category","digi")),  
4     TAURAMP_    (p.get<float>("TAURAMP", 1.4 /* ns */)),  
5     TAUDECAY_   (p.get<float>("TAUDECAY", 36.4 /* ns */)),  
6     PULSELENGTH_ (p.get<int>("PULSELENGTH", 30 /* samples */)),  
7     // ...
```

9.2 *Reading environment variables*

```
1 #include <cstdlib>  
2 //...  
3 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>.

```
1 #include "cetlib/exception.h"  
2 // ...  
3 if ( something ) {  
4     throw cet::exception(CATEGORY) << "Message\n"
```

```
5 }
```

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.

```
1  gm2util::MetadataFromFile::MetadataFromFile(
2      fhicl::ParameterSet const & p) :
3      searchPath_ (p.get<std::string>("searchPath", ".")),
4      fileName_   (p.get<std::string>("fileName")),
5      keyName_    (p.get<std::string>("keyName"))
6  {
7      // Let's parse the search path
8      // If the first character is a dollar sign, then the
9      // remaining is an environment variable
10     if ( searchPath_.at(0) == "$" ) {
11         std::string envVar = searchPath_.substr(1);
12         char* envValue = std::getenv(envVar.c_str());
13         if ( ! envValue ) {
14             searchPath_ = ".";
15             throw cet::exception("META_DATA_FROM_FILE") <<
16                 "Environment_ variable_" << envVar << "_is_not_set";
17         }
18
19         searchPath_ = std::string(envValue);
20     }
21 }
```

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 code¹ for a particular gm2 release is accessible in our release area for you to peruse. Set up the release (see section 3.1) and look in `$ART_DIR/source/art`.

My SSH login to gm2gpvmNN.fnal.gov dies sometimes. How do I keep from losing my shell environment and running processes?

The `screen` terminal multiplexer allows you to start a shell session which will persist (run in the background) even if the SSH connection terminates. Log in to the *g-2* Virtual Machine as usual, then type `screen` to start a new shell.² If your SSH connection is disconnected, then you can return to your shell by logging into the VM again and typing `screen -r` to 'reattach' to the shell.³ Screen sessions accept various keyboard shortcuts which indicate some terminal multiplexer action. Most are activated by pressing `Ctrl+a`, letting go of both keys, and then pressing another key within a couple of seconds which is tied to a specific command. For example, you can 'detach' from a `screen` session using `Ctrl+a` and then `d`, or you can start another shell using `Ctrl+a` and then `c`. The `screen` utility allows you to open many shell sessions simultaneously, and you can switch between them using `Ctrl+a` and a number which specifies the desired shell.⁴ Typing `exit` to close the shell session will drop you back to the previous shell, and typing `exit` in the final shell will exit the `screen` utility entirely, returning you to the original shell in which you ran the command `screen`.

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`.

² Note that you may have to run `. ~/.profile` in the `screen` shell session (though `~/.bashrc` should already be sourced for you).

³ If you have multiple instances of `screen` running, you may have to specify one by name (e.g. `12540.pts-4.gm2gpvm03`). The output of `screen -r` will list their names.

⁴ The shells are zero-indexed, meaning that you get to the first shell using `Ctrl+a` then `0`, the second shell using `Ctrl+a` then `1`, and so on.

11

Releases of gm2

This section describes the various releases of gm2. A description of the release philosophy is in Chapter 1. Some releases will have a `debug` build. You should only use those builds for debugging. Use the `prof` build for analyses.

The constituent versions of main packages are given for each gm2 release. You can see a list of all dependencies by running `ups depend`. For example,

```
ups depend gm2 v6_01_00 -q prof
```

For $g - 2$ products, you can usually see a change log in the product directory. For example,

```
less $GM2RINGSIM_DIR/CHANGELOG
```

Special migration instructions are given where necessary. Migration in general is covered in the developer workflow section.

11.1 gm2 v6_01_00 -q prof and (-q debug)

This is the first point release of the v6 series.

Contains:

- `gm2ringsim v3_00_00`
- `gm2geom v3_00_00`
- `gm2dataproducs v3_00_00`
- `artg4 v3_00_00`
- `gm2artexamples v3_00_00`

11.2 gm2 v6_00_00 -q prof and (-q debug)

This a base release of the v6 series. Note that you no longer have to specify the qualifier for the compiler version (is `e7` for this release)¹

¹ There is only one compiler version that works for this release.

As per the release philosophy, there are no g-2 packages in this release, only externals are released.

gm2 v6_00_00 has the following:

- `art v1_13_01` [Release Notes](#)
- `root v5_34_25` [Release Notes](#)
- `geant v4_9_6_p04a` [4.9.6](#), [p01](#), [p02](#), [p03](#), [p04](#)
- `gcc v4_9_2` with `-std=c++1y` for C++14 features.
- `gs1 v1_16` (GNU scientific library) (new!)

This release works on the following platforms:

- Scientific Linux 5
- Scientific Linux 6
- Mac OSX Mavericks
- Mac OSX Yosemite (new!)

Note that Mac OSX Mountain Lion is no longer supported.

There are significant improvements that speed up builds, including a replacement for `make` called `ninja`. See section ??.

11.2.1 How to migrate from v5_XX_XX to v6_00_00

Updating source code: The `develop` branches of the standard simulation packages are now compatible with v6_00_00. If you have a branch that you need to update, you can merge `develop` on to your branch with,

```
git merge develop
```

Be sure to read the next section for important changes.

Building with v6_00_00: In general, it is easiest to start with a new development area. You can re-use an old one by setting up the `g-2` environment, explicitly setting up this version of `gm2` and then, in the top level directory of your development area, do

```
mrbs newDev -p
```

That command will make a new `localProducts...` area. You must source the setup script in there to continue.

Missing symbols involving TFileService: The `TFileService` has changed and now involves a template, which means one must explicitly link in its library.

If you have a module that uses a `TFileService`, find the `CMakeLists.txt` file and add to the `art_make` after

MODULE_LIBRARIES, art_Framework_Services_Optional_TFileService_service.

For example,

```
# Fill Builder Filter Module
art_make( MODULE_LIBRARIES
          gm2analyses_calor_clustering
          art_Framework_Services_Optional_TFileService_service
        )

# Install header files into the products area
install_headers()

# Don't do clustering until we get Nic's CaloGeometry_service up and running again
add_subdirectory( clustering )
```

I have made this change for most of our packages in the feature/gm2v6 branch (now in develop).

Problems with Root on Mac Yosemite: The version of root setup by gm2 v6_00_00 (or any v6 series release) gives an error when you try to open a TBrowser (this only happens on the Mac Yosemite platform). An updated root for yosemite is available. When you are ready to analyze data with root, do the following beforehand:

```
setup root v5_34_25a -q e7:prof
```

This will give you a version of root that works on Yosemite.

11.3 *gm2 v5_01_00 -q e6:prof*

This is the first point release of the v5 series.

Contains:

- gm2ringsim v2_00_00
- gm2geom v2_00_00
- gm2dataproducs v2_00_00
- artg4 v2_00_00
- gm2artexamples v2_00_00

11.4 *gm2 v5_00_00 -q e6:prof and (-q e6:debug)*

Note the new version numbering scheme as per the release philosophy.

This release is the fifth one for g-2 since time began, thus the v5.

This release is the base release of the v5 series.

gm2 v5_00_00 has the following:

- [art v1_12_02 Release Notes](#)
- [root v5_34_21b Release Notes](#)
- [geant4 v4_9_6_p03e Release Notes: 4.9.6, p01, p02, p03](#)
- [gcc v4_9_1 with -std=c++1y for C++14 features.](#)

11.4.1 *How to migrate from v201402 to v5_00_00*

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. 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.

11.5 *gm2 v201402 -q e4:prof*

`gm2 v201402` has the following:

- [art v1_08_10 Release Notes](#)
- [root v5_34_12](#)
- [geant4 v4_9_6_p02](#)
- [gcc v4_8_1 with -std=c++11 for C++11 features.](#)
- [cmake v2_8_8](#)

and

- `gm2ringsim v1_00_00`
- `gm2geom v1_00_00`
- `gm2dataproducs v1_00_00`
- `artg4 v1_00_00`

This is the old release with the old date scheme. It should no longer be used.

11.6 *The Release Philosophy*

What is a `gm2` release? A `gm2` release is a versioned collection of libraries and executables that you either use or build your code against.

A particular `gm2` release contains a particular version of `gcc`, `art`, `root`, `geant4`, etc. These libraries/executables are called the *externals*. A `gm2` release may also contain $g - 2$ applications and libraries built against those externals (e.g. `gm2ringsim`, `artg4`). If the versions of those packages are suitable for you, then you can use them directly without having to build them yourself. This means we have *official* versions of these packages.

Official releases are important. For the purpose of scientific reproducibility, it is important to know how results were produced. Using a versioned release means that we know the code used for an analysis and can re-run it to do further analyses or look for mistakes. Official releases are essential for sharing code, as it gives people a common base and starting place.

The philosophy of `gm2` releases is that the first (major) version number in the release (e.g. the 5 in `v5_00_00`) is the release *series*. Releases in the same series are built with the same version of externals (`gcc`, `art`, `geant4`, `root`, etc) and so they are all binary compatible. The `vX_00_00` release only has externals in it and is called a *base* release. We then add point releases (e.g. `v5_01_00`, `v5_02_03`) containing $g - 2$ libraries and applications (e.g. `gm2ringsim`). If there is a new `art` or `root`, then the major version number increases (e.g. to `v6_00_00`) and a new series is started. New major releases (new series) should occur only 3-4 times per year.

The point releases can occur more often and represent official changes to the $g - 2$ code base (e.g. new geometry or features in the simulation). Feature changes advance the middle (minor) version number and bug fixes advance the last (patch) version number. So the first release of $g - 2$ code for a new series is `vX_01_00`. A feature addition will advance to `vX_02_00`. A subsequent bug fix will advance to `vX_02_01`.

Users adoption of these point releases is optional. They can always build all of the necessary code based on the `vX_00_00` base release. But using a point release can be convenient and save a large amount of time by using pre-built libraries instead of building them by hand. The point releases also represent a trackable official progression of features and bug fixes. Users can also use libraries from a point release, but build parts of the release themselves that they are developing (e.g. developing `gm2ringsim`, which is also in the release). In these cases, the build system will automatically use the user developed code instead of what is in the release. The `superbuild` system, which does builds across platforms for use on the grid, will automatically mark such user-built libraries as unofficial.

11.6.1 How do releases help me as a user/developer?

If the official released code is suitable for you (e.g. you are not developing the code in the release, but are instead developing code that *uses* the release), then using an official release will save you compilation time and will be more convenient. You can more easily track what you have run on the grid. You may be able to run without having to build anything (e.g. simply running the official `gm2ringsim`).

You should use an official point release whenever possible. Instructions for setting up your development area and how to migrate to new releases are given in the section under the release notes as well as in Chapter 3 of this manual.

12

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).

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.

12.1 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 v6_01_00`. On the bottom of the title page is the git version information for this document itself. For this version, it reads `v6_01_00_00-6-gb7d6c91-dirty`. There are three or four parts to this description, separated by *dashes* (not underscores;

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 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

12.2 Obtaining this documentation

The latest official version of this documentation is in GM2 DocDB as [GM2-DOC-1825](#).³ Newer releases of `gm2` (starting from `gm2 v5_01_00`) will have a copy of this manual that corresponds to the particular `gm2` version at `$GM2SWDOCS_DIR/manual.pdf`.

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

To get the source,⁴ follow the instructions in section ???. When you get to section ???, 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,

```
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.

12.3.1 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.

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

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

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.

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`.

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.

12.3.2 *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`.

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.

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.

Index

add_subdirectory, 35

art_make, 36

arguments, 37

artmod, 35

CMakeLists.txt

directory level, 36

top level, 33

exceptions, 45

external code, 38

find_ups_geant4, 41

find_ups_product, 39

find_ups_root, 40

input source

writing, 36

install_headers, 36

linking, 38

modules

writing, 35

services

writing, 36