ResoNEUT Analysis User Manual DRAFT

Sean Kuvin

February 7, 2014

1 How to Get the RNAnalysis Packages

1.1 Git

Git is a distributed revision control system. If you are familiar with Subversion or other version control software, learning how to use git should not be difficult. Regardless, there are plenty of tutorials online for developers with varying levels of experience. I recommend these links to get started:

- BitBucket 101
- Git for the Lazy
- Simple Git WorkFlow

1.2 ROOT

An installation of ROOT is also required. The code has been tested on ROOT Version 5.34. To properly compile the resoneut_analysis code you must make sure to source the thisroot.sh shell script included with the version of ROOT you installed. Do:

```
. path/to/root/bin/thisroot.sh
```

I recommend adding this line to your .bashrc (if you are using bash) file so that ROOT is automatically setup everytime you open a new terminal. If you forget to do this the RNAnalysis code will not compile. The first error you will see, if you forget this, is something like "RConfig.h was not found."

1.3 Getting Started

First thing is to make sure you have git installed. If you do not have git, get it by going to the link provided. If you do not have sudo access, you can compile the code yourself from the git source repository.

Once you have git, create a folder which will contain the repository. Then clone the repository using the FSUSER login

```
mkdir MyRepoDirectory
```

```
cd MyRepoDirectory
```

```
git clone https://FSUSER@bitbucket.org/skuvin/resoneut_analysis_hc.git
```

FSUSER has only read permissions. So trying to push any changes you have made back to the main repository will be futile.

You should now have the resoneut_analysis_hc repository in your directory.

1.4 Installing

If ROOT is properly installed and thisroot.sh has been sourced then you can try compiling the code. There is a very unsophisticated recursive Makefile in the top level of the directory. It will enter the include folder and each plugin folder and execute the Makefile in each of those. Before doing this, if there is no bin folder or lib folder in this directory you should make one of each.

```
mkdir bin;mkdir lib
```

Executing Make in the include folder will create two programs, RNBatchMode and RN_SimRun, and place them in the bin folder. It will also create the libRNeut.so shared library in the lib folder. This shared library is loaded into a running instance of ROOT in order to interactively use the RNAnalysis packages. Executing the make in the plugins folders will add more shared libraries to the lib folder.

NOTE: Your installation of ROOT, OS, or gcc compiler, may require minor tweaking of the Makefiles in order to get them to work. As I encounter common issues I will record them at the end of this section.

1.5 Setting up

RN_BatchMode is used to convert .evt files to .root files. RN_SimRun is the kinematics simulator. Both of these programs should already be working, given a proper configuration file. Examples are provided in the examples folder.

In order to use the RNAnalysis packages to further analyze those new root files you need to load the shared libraries into root. To do this you must first run root:

```
root -l
```

then, while in root do:

```
gSystem→Load("path/to/lib/libRNeut.so")
```

you can test to make sure the classes are properly loaded by creating an instance of an RN_S2Detector

```
RN_S2Detector a("si_a",16,16)
a.front.NumOfCh()
a.InsertHit(10,5,2)
a.front.fMult
a.Front_E()
```

I recommend using a rootlogon.C script to automatically setup the framework. if you run root -l when there is a rootlogon.C script in the current directory it will automatically execute the commands in that script. A rootlogon.C example is provided.

1.6 Known Compiling issues

- When compiling on a Mac, need to add -L$(PWD)/../../lib -lRNeut -lSAK to the libRN-analyzer Makefile.
- ROOT v3.2 sometimes has an extra directory inside of the include folder (ie /root/include/root/) before you get to the .h files. in v3.4 this include folder looks like root/include/*.h files. If you encounter this problem you need to make sure the make dependencies include the proper directory.

2 Unpack To Root – RNBatchMode

2.1 Goals

The RNAnalysis packages rely heavily on the ROOT libraries and framework. In order to take full advantage of the code that has been written we must first convert the binary .evt files into a simple TTree structure.

2.2 NSCL Ring Buffer

The .evt file is formatted based on the new NSCL RINGMASTER format. With RINGMASTER, each buffer is of a variable length in bytes depending on the number of modules present in the configuration stack and of the number of "hits" in each module. With zero suppression, a hit would typically be a channel or channels with a non-zero value. Depending on the number of these channels, the length of the buffer will increase or decrease.

2.3 Setup

The Unpack2Root method consists of multiple functions encapsulated in the namespace "unpacker". Within this scope, there are global variables which define the types of values which are to be stored within a TTree and then written to a root file. Furthermore, when it is time to read from these root files it is simple enough to pass the stored data back to these same global variable addresses.

Each module is stored as an array of Float precision variables (i.e. ADC1[32]) where the 32 corresponds to the maximum number of channels in the detector.

In order to add a new module the following steps must be taken:

- create a global variable for the new module ((i.e. Float ADC8[32])
- set the variable as extern in the .hpp file so it can be made global in other pieces of code which include the RNUnpacker.hpp header file.
- Add it to the ResetTreeParameters() function so that all members of the array are reset to zero. This function is called before each buffer is extracted from the .evt file
- Add it to the SortGeoChan() function so that the geoaddress matches with the proper module.

- In Conver2Root, add the module to the data tree by setting one of the tree branches to the location of the new module's global variable. This is done by DataTree→Branch("ADC8",&ADC8,"ADC8[32]/F
  Whenever DataTree→Fill() is called, all Branches which are set by this method will be written to the tree.

- The final step is to add the newly created module to the configuration file(example: e2rconfig.in). This is done by adding the appropriate geoaddress number to the caen stack or the mesytec stack.

The e2rconfig file must have stacks which match the stacks from the daqconfig file set during the data acquisition. If a module needs to be removed, just remove it from this configuration file. NOTE: the code checks to make sure that each buffer follows the proper order of the stacks. If the e2rconfig stack does not match the daqconfig stack for that particular run then the resulting rootfile will be corrupted.

The order is set so that caen modules are placed in the VMUSB crate first, then the mesytec modules. NOTE: This ordering is hard coded into the UnpackPhysicsEvent() function.

Other information can also be written to this TTree. As of this writing, the Event[3] array stores the run number, an error flag, and a scaler flag. If a composite rootfile consisting of multiple runs is created, it is still possible to extract which runs correspond to relevant data.

The Scalers from the event can also be written to the ScalerTree by adding the appropriate scaler stack to the e2rconfig. If a scaler buffer exists in the .evt file, by ring buffer default, it will be a fixed size consisting of all scaler channels. In order to read out the scalers in order, add a new entry to the scaler stack by providing a name to that scaler channel. If three names are entered, seperated by whitespace, then the first three scaler channels will be written to the ScalerTree. Each time a scaler is written to the ScalerTree, a scaler index is incremented. This scaler index is stored in DataTree-¿Event. From this index, we know what data corresponds to a particular scaler interval.

2.4 RNBatchMode

The work horse of this unpack2root method. the RNBatchMode.cpp has a main which is compiled into a standalone program. It looks for three additional arguments.

RNBatchMode <Data Directory or evt file> <output rootfile name> <e2rconfig file appropriate for this set of runs>

If a .evt file is provided then the code will immediately produce a root file with the given output name.

If a data directory CONTAINING the .evt file is given, then an interactive prompt will be provided asking which MULTIPLE .evt files should be extracted into a SINGLE output root file. if the evt file has the name run-2401-00.evt the run number should be provided with the format 2401-00.

3 RNAnalyzer

3.1 Goals

The RN_Analyzer class is meant to read .root files that were created with RNBatchMode. An analyzer is first initialized using Init(rootfile). More root files can be added by doing AddTree(rootfile). The way the class reads from the tree is based on the simple MakeSelector structure.

The goal is to have a generic RN_Analyzer which extracts the raw module data and passes it to the appropriate detector. Besides looping over all the events in the Chain of trees that were added, this is all it does. There will be, however, derived classes which inherit from RN_Analyzer. These classes will overwrite the Begin(), ResetGlobals(), Process(), ProcessFill(), and Terminate() virtual functions. By having a list of derived Analyzer classes each with their own unique functions, we can loop over all created analyzers and call EACH of their functions at the appropriate time. For example, all derived analyzers will call their own Begin() before looping over events.

3.2 MainAnalyzer

In the RN_Root.cpp file there is a global instance object of RN_Analyzer called MainAnalyzer. To set it up we create an .C script (an example is provided in the examples folder as analyzerscript.C)

We can set up a simple analysis by adding the following lines to the .C script:

```
{
 MainAnalyzer.Init("run2021.root")
MainAnalyzer.Loop()
}
```

and then executing the script by doing (with root running):

```
.x analyzerscript.C
```

The MainAnalyzer will Loop() over all of the events in run2021. During the Loop() process a few things happen:

- Before looping over the events, the Begin() function is called. This prints the total number of entries in the Chain.
- During the loop, for each event, ResetGlobals() will be called and then it will unpack the ADC or TDC modules into the appropriate detectors using the GetDetectorEntry() function.

- Some important quantities will be calculated in the Process() function
- ProcessFill() is called, which for a generic RN_Analyzer does nothing.
- Then the loop continues, going to the next event
- Finally, after the loop is over, the Terminate() function is called which also does nothing for a generic RN_Analyzer.

So by executing MainAnalyzer.Loop() we extract the root file data into the detectors but since ProcessFill() does not do anything the Loop finishes and we do not see any result. If you notice, however, in the RN_Analyzer::Loop() function, there are while functions which loop through something called a TList. This TList contains analyzer instance objects which have been added to said list. These Analyzers inherit from RN_Analyzer and overwrite the virtual functions Begin(),ResetGlobals(),Process(),ProcessFill(), and Terminate(). This means that at each point when MainAnalyzer calls one of these functions, it will also call the same function in each of those Analyzers.

For instance, say we create an object of Si_Analyzer (in plugins/analyzers) andit is added to "analyzers" (the TList). When MainAnalyzer.Loop() is called, not only will MainAnalyzer's Begin() be called but also Si_Analyzer's Begin(). When MainAnalyzer's Process() is called so will Si_Analyzers'. All Process()s will be called before ProcessFill()s

This is the main point about the analyzer list. If ANY analyzer which is added to the list returns a 0 during the Process() function, then the MainAnalyzer will abort that event and skip ALL of the the ProcessFill()s. ProcessFill() is where we fill histograms or trees so therefore, we can filter through our root file immediately by just returning 0 if a condition is not met in a particular analyzer's Process().

There is also a TList called analyzerlist which allows the user to create a list of all of the Analyzer instance objects that are created.

4 Global Variables and Functions for RNRoot

4.1 Globals / Main Functions

The RN_Root.cpp file contains the basic setup for the ResoNeut framework. It contains the global variables and objects that are used for the data analysis. A summary of the main functions are provided below:

RN_RootInit() provides the initial setup of the Detectors including how many of each type and an identifying name.

LoadVariableFile("config_file.in") imports a text file and adds it to the RN_VariableMap DetVar. An RN_VariableMap is simply a std::map which maps a double precision variable to a string.

SetCalibrations() takes that DetVar VariableMap and passes it to all of the Detectors. Each detector has a function of its own called SetCalibrations(RN_Variable_Map) which searches through that map for a string which starts with its own name and ends with a variable name which it knows about. For instance, if a detector is called "si_a" and the channel map contains an entry with the string "si_a.elin" then "si_a" will take the value associated with that string and set the elin member variable to that value. To find out what calibration strings a detector knows about, check the SetCalibrations() function associated with that detector.

SetRootOutputFile(string filename) sets the rootfile name that will be the output file for all of the histograms created in the Analyzers.

SetRootOutputFileAndTree(filename,treename) is necessary if you plan on creating a new tree structure and want to fill it in the same root file with the histograms.

global::SetReaction(a,b,c,d,e,f) sets the masses and names for the reaction particles. The masses are gotten from the RN_MassTable, and are looked up by the names provided. If you want to add a new mass, add them to this table.

AddAnalyzer(TObject *obj) adds an Analyzer instance to the list of Analyzers that MainAnalyzer knows about. See the section on RNAnalyzer for more information.

5 How To: A Simple Filter Method

5.1 New Tree Analyzer

In order to create a filter file, use the NewTree_Analyzer. First you must use the SetRootOutput-FileAndTree(filename, "DataTree") instead of the SetRootOuputFile(filename) in your analysis script. Then create an instance of NewTree_Analyzer and add it the analyzer list.

The new tree will only be filled if the Analyzer list makes it to the ProcessFill() sequence. This means that if you put any filters in the Process() portion then you will automatically create a filtered root file.

For example, in the Si_Analyzer there is a filter called RequireProton(). If this filter is turned on in the analysis script, then the event will be aborted whenever an event does not have a silicon E-dE which fits the proton gate, prots1. This will then result in a new root file with a filtered tree.

By filling a new tree with the same format (and tree name: "DataTree") we can then rerun this new filtered tree with the same analyzer scripts.

6 Silicon Detector Methods

6.1 Auto Calibrate

6.2 Reconstruct Clusters