# SPS Digital Event Builder v.2

Gordon McCann
Contributions from: Ken Hanselman, Erin Good, Sudarsan Balakrishnan, Kevin Macon

## 1 Overview

This paper is aimed at giving a brief example of how to use the EventBuilder code. Note that this code is a *template*. I will outline a general method by which the event builder will work, but most experiments will require the user to tweak the code to fit their individual data set. Additionally I will try to give pointers on how the user can modify the code to better fit other use cases. Hopefully this will allow everyone to work on SPS data from the same basic template!

Some notes about compatibility: The code has been compiled successfully on Scientific Linux 7 and Mac OS X Catalina. The code is currently incompatible with Windows, and there have been issues with linking ROOT to some versions of Ubuntu. Testing has been done with ROOT ver. 6.14 and 6.20, so anything else is a toss up.

Other than that, I'll get started with how to build the code.

## 2 Building

As of version 2, there is no longer any dependency on `libarchive` (hooray!), which will hopefully save everyone some headaches.

Building is handled via the `make` command. To build the entire environment simply enter `make` into the command line and watch the output. To build from scratch after an initial build run `make clean` as this will delete all objects and executables, except for one special case. The build uses a precompiled header which contains the commonly used standard library and ROOT library includes (EventBuild.h). This is typically built only once, during the first call of `make` and never needs to be touched again. However, if for some reason you need to clean this as well you can use `make clean_header` to delete the file and force a rebuild.

Note: This makefile is extremely general. Any time a file with a `.cpp` extension is added into one of the `src` program directories it will try to build it. If the user wants to add and additional program/directory, just follow the format outlined here.

## 3 Data Structure and ROOT dictionary

Now is probably a good time to get into how the data is structured and given to the ROOT environment. To find the data structures, simply open the file `DataStructs.h` in `include`. In here you'll see several defined `C` structs, namely:

- `DPPChannel`
- `DetectorHit`
- `SabreDetector`
- `FPDetector`
- `CoincEvent`
- `ProcessedEvent`

`DPPChannel` is a modified structure of the standard CoMPASS data. The modification is in the typing of the data; CoMPASS reports out in terms of unsigned integers, which can be difficult to manipulate with mathematic operations. Early in the event building, these unsigned quantities are converted to signed counter parts of appropriate length.

`DetectorHit` carries the relevant information of any detector hit: energy (`Long`), timestamp (`Time`), and channel (`Ch`). Note that we use here only the long gated energy; if there is an application which requires using the short gated energy, that should be added into this structure. There are then two detector structures shown here: `SabreDetector` and `FPDetector`. These structures contain combinations of detector hits that are organized by components for each detector array. For example `SabreDetector` has elements for rings and wedges, while the `FPDetector` has elements for the anodes, delay lines, scintillators, and cathode. Note how these are all `std::vector` type quantities. This is important for the way that event building is done. All detector elements should be initialized as `std::vector`s unless you have a very good reason to assume that this detector element will only ever seen ONE hit (including noise) during a coincidence window.

`CoincEvent` is a collection of detectors that should be grouped together in coincidence. This is the quantity which we desire to have the event builder create for us! Here I have an example with an `FPDetector` and an array of 5 `SabreDetectors` (one for each silicon in the physical array).

`ProcessedEvent` is the physics output of the event builder. This is where you would want to define quantities you actually use in real physics analysis. For example, in my version I have defined focal plane positions, angles, and various energies and times. As these are the physical variables, they deserve a little more detail, given below:

- `fp1_tdiff` and `fp2_tdiff` are the time difference in $left - right$ format of either delay line. These are in nanoseconds.

- `fp1_tsum` and `fp2_tsum` are the sum times $left + right$ of the delay lines (ns). This sum should be "consistent" with the total delay of the delay line.

- `fp1_tcheck` and `fp2_tcheck` are the sum times divided by 2, with the anode time subtracted. Essentially, this tells the inherent time resolution of the delay line scheme. There are also some other uses for this parameter, depending on how much you trust the anode time.

- `fp1_y` and `fp2_y` are basically the drift time of each wire, which means they are a measurement of the y position. It should be noted, that after some testing these were found to be of such poor resolution that they were not useful.

- `anodeFront/Back`, `scintLeft/Right`, and `cathode` are all energy values of their respective components.

- `x1` and `x2` are the `tdiff` parameters converted into mm. They are the delay line positions.

- `xavg` is a weighted average of `x1` and `x2` with weights calculated such that `xavg` is the result of a linear interpolation (or extrapolation depending on the scenario) to the kinematic focal plane. If you drew a line with `x1` and `x2`, `xavg` is where that line intersects with the kinematic focal plane.

- `theta` is the incident angle of the particle, calculated by $arctan(\frac{x1-x2}{\text{wire separation}})$

- `sabreRing/Wedge`, `sabreRing/WedgeChannel`, and `sabreRing/WedgeTime` are SABRE data for the largest energy hit in each SABRE detector (one for each detector). These are easy access values for the cleaner.

- `dealyFront/BackLeft/RightE` are energy values from the delay lines themselves. It has been found that in many cases these have higher resolution and less deformation than the anode signals.

- `anodeFront/BackTime` and `scintLeft/RightTime` are the times of the different components (ns).

- `delayFront/BackMaxTime` are the maximum time of the left and right side of the delay lines. These are very useful for estimating the width of the slow coincidence window, as once everything is shifted into place, the delay time is the only remaining time that needs to be accounted for in the window, and even that only needs to be the *maximum* delay.

- `sabreArray` is the full set of SABRE data for the user to handle in the next level of analysis.

In general, the user will be making the most changes to what a `CoincEvent` is and what a `ProcessedEvent` is. These are the parts that really depend on the specifics of each experiment. It should also be noted that for many of these I initialize the structure to unphysical values (negative numbers for energy and time, very large numbers for position). The reason for this is that ROOT TTree fills are not, in general, selective, while most analysis codes are. You define an address for a branch, and then every time you call `MyTree->Fill()` the tree adds an entry to each branch *regardless* of whether or not the value in that branch address has been updated. This means that to make the fill safe, we need to reset the value at the branch address *every time* we go to perform an assignment. This means that if you add member to something like `ProcessedEvent` you should set it to something unphysical which can be easily discarded at a later stage (I will refer to these as dump values).

Now that our data structures are defined in our code, we need to define them for the ROOT environment so we can use them with our TTrees. This is done via ROOT dictionaries. If you are looking for a really detailed description of how this works, I suggest going to the ROOT user guide for more information, as there is a lot of black box type behavior here. ROOT installations come with a dictionary generator `rootcling` which does pretty much all of the heavy lifting for us. To use `rootcling` we need to have a file that defines which structures need to be added to the dictionary. This is called the `LinkDef.h` file, and for us this file is called `LinkDef_sps.h` (`rootcling` can only accept header files as arguments). The `LinkDef` file contains a whole bunch of preprocessor directives. First it checks to make sure cling is well defined, and then it links all of the structures we want. Note that here I've used structures because our data is pretty simple, but you can link more complicated things like classes if you need. The building of the actual dictionary is handled by the makefile.

Additionally, if one wants to use data made in such a fashion outside of the `EventBuilder` environment, the dictionary will need to be loaded/linked into that space as well. To this end, a dynamic library is generated in the `lib` directory, along with a copy of the `.pcm` file. To use the dynamic library in by linking in another executable, either add the `lib` directory of the `EventBuilder` to the library search path and link as normal, or use a method similar to the `libarchive` linking method shown here. Additionally, the `.pcm` file will need to be copied to the location of where ever the new program is. If you would like to use the dictionary in a ROOT macro, move the dynamic library and `.pcm` file to the same directory as the macro, and in the macro add the line

```
R__LOAD_LIBRARY(libSPSDict.<suffix_for_your_os>)
```

right after your `#include`s and before your macro function. Note that in both cases you will also still need to `#include "DataStructs.h"` to actually use the structures, so this header will also need to be moved/added to the include path to give full functionality.

## 3.1 Additional notes on dynamic libraries

Above I outlined how one would use the auto generated dynamic library for the dictionary to extend the usefulness of data generated in this environment. There is a bit more information that needs to be passed on about how this is done currently, how you can do this for other classes in the `EventBuilder`, and future plans.

First, dynamic libraries are OS dependent. Currently, I have the `makefile` check your OS for either `Darwin` or `Linux`. This means there is no support currently for Windows machines, since I don't know how they generate dynamic libraries or how to check their OS. If you're running MacOS (`Darwin`) the `g++` option `-dynamiclib` is passed and a `.dylib` file is generated. If you're on `Linux`, the `g++` option `-shared` is passed and a `.so` file is generated. This is because on MacOS shared libraries and dynamic libraries are not necessarily the same (I think, I'm not an expert on this). Either way, one of these two options is passed and a dynamic library is generated *from* the static object. Additionally, there may be other classes than just the dictionary that you'd like to incorporate into a ROOT macro or some other code. To generate a dynamic library from the `analyzer` program (the only one with features that could actually be used elsewhere), use the following make command:

```
MacOSX: make lib/lib<name_of_cpp_file>.dylib

Linux: make lib/lib<name_of_cpp_file>.so
```

A dynamic library should be generated into the `lib` directory. Note that the dictionary is the only one that needs a `.pcm` file as this is unique to ROOT dictionaries.

## 3.2 Example of dynamic library in action

Here is an example of a ROOT macro which uses a dynamic library as if it were in the top directory of the event builder, along with the associated header file:

```
#include <TROOT.h>
#include "include/DataStructs.h"

R__LOAD_LIBRARY(lib/libSPSDict.dylib)

void this_is_a_test() {
  DetectorHit* h = new DetectorHit();
  tree->SetBranchAddress("hit", &hit);

}
```

Normally if you tried to run this macro without the `R__LOAD_LIBRARY` you would get an error saying that DetectorHit is not defined properly for use with TTree. With the load in place, now ROOT has all of the info it needs to make this happen.

## 4 The Event Builder Program

If you've used the Event Builder in the original version, you'll notice that a lot has changed! The focus of the revision was towards trying to blend easier use with better performance, and I hope that at least some of that was actually achieved. I'll outline the event building process, and emphasize some of the changes from the previous version. The order in which I describe the code is the order in which pieces are expected to be used.

### 4.1 binary2ROOT conversion

In the previous event builder, conversion of data from a raw CoMPASS binary file was simple and only transferred data from the CoMPASS format to a ROOT format. However, there were several flaws to this implementation that led to headaches. First and foremost, was that information was actually discarded. The previous version essentially stitched together the raw binaries (each binary file contains the entire data for a single digitizer channel). This basically discards the fact that data within these binary files *must already be sorted in time*! Since time ordering data proved to be one of the largest overheads of the original version, this seemed a good place to start for increasing performance. A complete overhaul of this process resulted in the current conversion methods, which are written in the files in the `/src/binary2root/` directory. The basic concept is the following: we open up a data stream for each individual file, query the files for hits, and then write out the most recent hit in the list of hits we get. The file which contained the used hit is then queried for its next hit, and the process continues until the data in all files is exhausted. This requires that timestamp shifts be applied at this stage of the program (this is a change from the previous version which shifts were applied during the bulk event building), which is handled through a new format for the ShiftMap. Now all shifts are specified in a single file, which reduces chance of confusing different shifts. Most importantly, this gives the user direct control over the amount of memory allocated per file for buffering; the previous method relied on ROOT functionality, which proved to be unpredictable when combined with large data files. One other change is the discarding of `libarchive` methods. `libarchive` requires that the archive be read sequentially, which would make the simultaneous opening of all binaries impossible (at least as I understand it). This means that the event builder no longer requires any linking to `libarchive`. This does come at a cost though: the archives must then be unpacked and decompressed at execution to a temporary location. Currently this is handled through system calls within the program. It will unpack the data to a directory in the ROOT directory called `\temp_binary\`; if this directory does not exist the program will fail. See the example for more details on this. After conversion this temp directory will be wiped again using a system call, so **SO DONT PUT ANYTHING IMPORTANT IN THE TEMP DIR**.

## 4.2 Event building

Since data are now already time ordered, the event builder solely focuses on creating events. As in the past, there are two levels of event building, slow and fast. Slow event building is handled by the SlowSort class; the actual sorting unchanged from the previous version, but for new users the details are as follows: data (referred to as a hit) in the raw ROOT file is organized into chunks called events by examining wether or not an event falls within the slow coincidence window. The start of the window is defined by the first hit (regardless of what type of hit it is), and once a hit is found that does not fall within the coincidence window, this list of hits is processed and assigned to detector structures for higher level analysis. The next event is then started by the hit in hand, and the process continues. The main change is in the method by which hits are assigned to detector structures. Previously this was done rather messily by using large conditional statements (typically a combination of `switch` and `if`) and could result in a large performance penalty, as well as being generally difficult to debug. To improve this, the ChannelMap class was created. This class wraps a map which is defined in a text file (an example is included); the format is similar to what was previously referred to as the SabreMap, however now it is used for *every* digitizer channel. The user then defines integer keys which uniquely map the global channel number to a specific detector variable pointer. In this way it is somewhat similar to a tree concept. This allows the number of control statements to be reduced significantly. Basically the code only needs to query two maps to assign data to the correct detector structure, rather that using a vast control block. More details on the specifics of modifying the ChannelMap are given in the example section.

Fast sorting is completely unchanged in process from the previous version. Again, this is very user specific, so it is up to the individual to ensure that this piece of the pipeline meets their specific needs in terms of efficiency and readability. For new users, the FastSort class is used as second level of event building designed to resolve any discrepancies in the slow event. For example, there are two scintillator events with in the length of the slow window, yet only one ion chamber (anode, delay line, etc). Which scintillator is actually correlated with the ion chamber data? The goal of the fast sort is to resolve such situations, usually by providing a secondary coincidence window for two specific detector components.

Finally, the event builder also has a section for basic analysis. No changes were made from the previous version. This is intended to be a location where the user can convert detector data into the physics parameters they would use to judge whether or not the event builder was successful. This IS NOT intended to be the location where the entirety of the data analysis occurs! It can certainly be used that way, however it will slow down the event building pipeline, which will make it more time consuming to dial in the event building parameters and get the data to a good event built state.

## 4.3 Plotting

The event builder also contains functionality for generating a histograms with externally defined gates and cuts. Again, this is very experiment specific, so this will most likely need to be modified on a user to user basis. The current version has been update to read a text file to get cut file names, rather than taking them as input arguments. This simplifies the input and output, and makes it easier to modify.

## 4.4 Merging

The event builder also contains methods for merging a set of ROOT files with **identical** trees into a single file. This is intended to be used once the user is satisfied with the event built data, to combine all runs into a single file for use in the next stage of analysis.

## 4.5 Archiving

The event builder has methods for archiving and compressing raw binary files from CoMPASS. This is not intended for normal use; it is meant to be used within the GUI, which is still under development.

## 4.6 GWMEventBuilder API

One of the more significant changes from the original version, is the addition of an API class named `GWMEventBuilder`. The intention is to simplify user modification to event building, reduce the number of executables, and pave the way for a GUI. Basically anything that was in a `main` function before, is now a method in `GWMEventBuilder`. This also reduces the number of input files to one single input. The hope is that this reduces some of the more frustrating *quirks* of the original event builder, and makes it easier to tailor the event building process to specific applications.

# 5 Example

Lets look at an example of what running this would actually look like. In the repository I've included a directory called `example` which contains both this document and an example run called `run_75`. This is $^{12}$C($^{3}$He, $\alpha$)$^{11}$C data from an experiment that was run in March 2020.

Let's set our input file up. Open `input.txt` with your favorite text editor, and you should see the following:

```
-------Data Locations----------
ROOTDirectory: /Volumes/LaCie/9BMarch2020/
BinaryArchiveDirectory: /Volumes/LaCie/9BMarch2020/raw_binary/
BinaryFileDirectory: none
```

```
MergeInputDirectory: none
MergeOutputFile: none
PlotOutputFile: /Volumes/LaCie/9BMarch2020/test_run99_v2.root
-------------------------------
------Experimental Inputs------
ChannelMapFile: ./etc/ChannelMap_March2020_newFormat_092020.txt
CutListFile: ./etc/CutList_March2020.txt
ZT: 5
AT: 10
ZP: 2
AP: 3
ZE: 2
AE: 4
BField(G): 9511.0
BeamKE(MeV): 24
Theta(deg): 20
-------------------------------
-------Timing Information------
BoardOffsetFile: ./etc/ShiftMap_April2020_newFormat_10102020.txt
SlowCoincidenceWindow(ps): 1.5e6
FastCoincidenceWindow_IonCh(ps): 0.125e6
FastCoincidenceWindow_SABRE(ps): 0.25e6
-------------------------------
--------Run Information--------
MinRun: 99
MaxRun: 99
-------------------------------
```

These are the settings I used recently to build the data from the same campaign our run 75 came from. So modify the directories to point to the example dir, and set the kinematic parameters to match the reaction of our data as shown below (`<your_path>` is a place holder):

```
-------Data Locations----------
ROOTDirectory: <your_path>/GWM_EventBuilder/example/
BinaryArchiveDirectory: <your_path>/GWM_EventBuilder/example/raw_binary/
BinaryFileDirectory: none
MergeInputDirectory: none
MergeOutputFile: none
PlotOutputFile: <your_path>/GWM_EventBuilder/example/histograms/
   run_75_histograms.root
-------------------------------
------Experimental Inputs------
ChannelMapFile: ./etc/ChannelMap_March2020_newFormat_092020.txt
CutListFile: ./etc/CutList_March2020.txt
ZT: 6
AT: 12
ZP: 2
AP: 3
ZE: 2
```

```
AE: 4
BField(G): 9511.0
BeamKE(MeV): 24
Theta(deg): 20
-------------------------------
-------Timing Information------
BoardOffsetFile: none
SlowCoincidenceWindow(ps): 3.0e6
FastCoincidenceWindow_IonCh(ps): 0.125e6
FastCoincidenceWindow_SABRE(ps): 0.25e6
-------------------------------
--------Run Information--------
MinRun: 75
MaxRun: 75
-------------------------------
```

We'll leave the default ChannelMapFile (its the one we want to use anyways), set our board offset to none at the moment, and widen our slow window since we don't know the correct values yet. Run the program with the following commands in the terminal:

```
./bin/GWMEVB_CL convert input.txt
./bin/GWMEVB_CL buildSlow input.txt
```

The first command will convert our data into ROOT, performing timestamp sorting and shifting. The second will build slow events and perform the basic analysis. Open the analyzed file and lets make some plots. There are default made histograms, but these aren't always best to use. First let's do particle ID through E-dE. Run the following command to the ROOT interpreter with the analyzed file open:

```
root [2] SPSTree->Draw("delayBackRightE:scintLeft>>TH2F(512,0,4096,512,0,4096)","","colz")
```

and you should see the plot shown in Fig 2. Now lets gate on the $\alpha$s, as shown in Fig 3. Since we're not done with event building rough cuts at this stage are fine. Now you can save this cut to a file by right clicking it and selecting the option `SaveAs`. Give its file a name, and then move the file to `./example/cuts/`. Below are example plots (Fig's 4) for all of the other cuts in the default code.

Ok so now that we've made our cuts, modify the file `/etc/CutList_March2020.txt` to reflect the names of your cuts and paths. Now you can run the plotting so that we can determine the shifts necessary to line up our data through the command:

```
./bin/GWMEVB_CL plot input.txt
```

Now if we open up our histogram file, we can begin to determine our shifts. First we'll determine the shift fo the scintillator to line it up with the back anode. Open up the plot labeled `anodeRelBackTime_toScint` and you should see the following plot Fig 5 We want to center the big peaked structure around 0 (anode back time minus scint left time is 0).

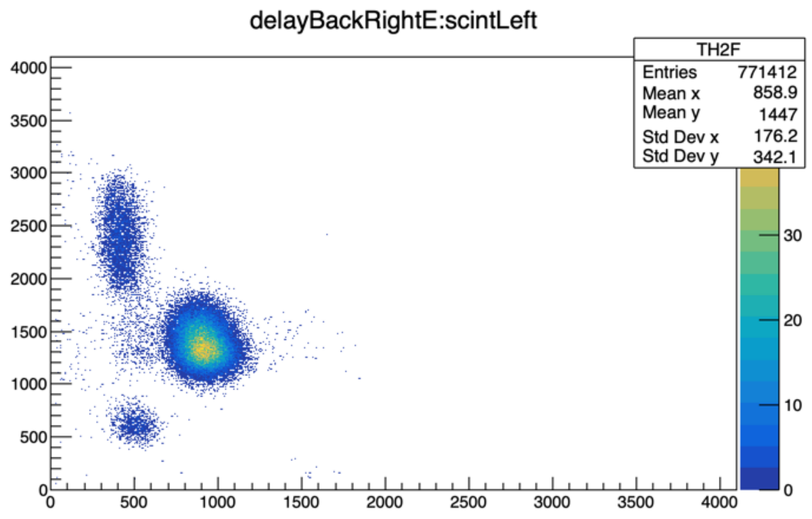


Figure 1: Back right delay energy (dE) vs. left scintillator energy (E), one of the common particle ID plots

Its clear that we need to shift the scint about 0.65 $\mu$s. Now, this experiment used SABRE, but $^{11}$C is stable for our energy range, so there is very little data in SABRE. If you were to look for SABRE shifts you would use the plots `sabreRelRTScint_sabreRingChannel` and `sabreRelWTScint_sabreWedgeChannel` to determine board by board shifts. I've included the plots (Fig 6) from this data set as an example, and we'll shift them by the shifts that I know apply to this data set anyways so you can get the picture. Note that in this method SABRE shifts are determined relative to scintillator shifts! This means that the shift you see that needs applied in the above SABRE plots needs to have the scint shift subtracted from it for you to use it. Before we move on, lets take one look at what our final data looks like in `xavg_bothplanes_edecut` and `xavg_edecut_sabrefcoinc` histogram (Fig. 7). It is evident that even with bad windows and no shifts, for simple data like this the event builder does reasonably well. Our states are there and sharp, but there are clearly fake coincidences being built with SABRE. We should be able to cut down on these.So now you can modify the shift file in `/etc/` to reflect the values you found (or just use the default one, its correct for this data) and add its name to the input file. Now since we modified the shifts, we need to run the entire pipeline again: convert, buildSlow, plot. Now open up the histogram file and your `anodeBackTime_toScint` histogram and you should see Fig. 8. Now lets also look at a slightly different histogram: `delayRelFrontTime_toScint`. This is a histogram of the maximum delay time for the front wire relative to the scintillator, shown in Fig. /refdelayRel. This shows us how wide our "slow" coincidence window should be! The delay lines are by far the slowest portion of the data set, and since everyone else is

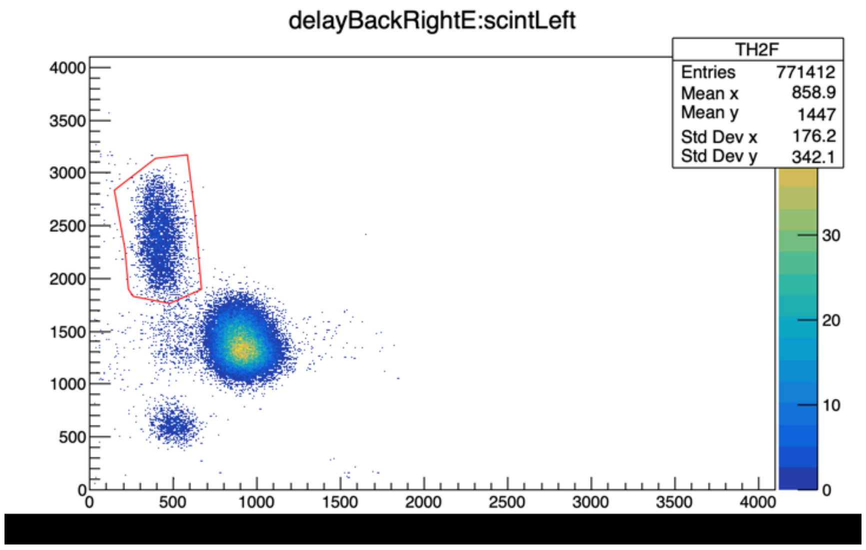


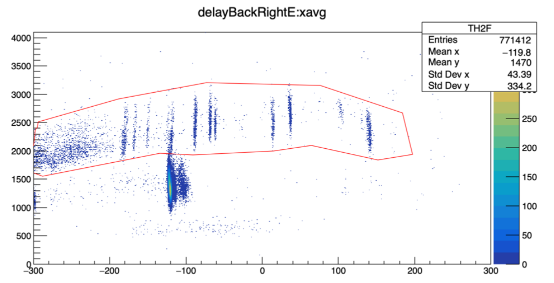
Figure 2: E-dE plot with a gate on $\alpha$ particles

centered at the same time stamp, we only next to extend far enough to catch the maximum delay. NOTE: you also need to account for the width of the centered peaks (i.e. the width fo the anode relative to scint peaks) in this window as well. Typically this is just adding a small extension onto the window to the delay time. For this data set 1.5 $\mu$s is good for a slow window. For our fast window, we can use the width of the anode relative to the scintillator peak to make our condition. This condition basically says that if the anode relative time doesn't fall within this window, it is not a good focal plane event! We can do the same thing for the SABRE relative to the scintillator time, using the default values here for this data since there isn't really much to see with a stable residual nucleus. Modify these values in the input file. Now lets run it one last time, with the following commands:

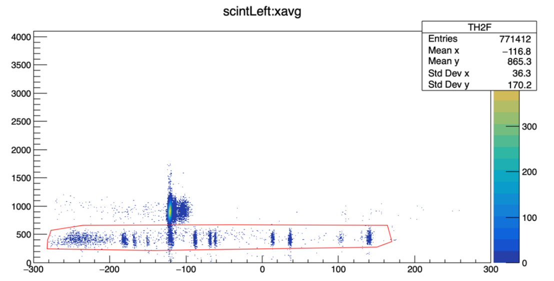
```
./bin/GWMEVB_CL buildAll input.txt
./bin/GWMEVB_CL plot input.txt
```

and see what we get. If you open up your histogram file and look at the same xavg plots from before you should see something like Fig 10.

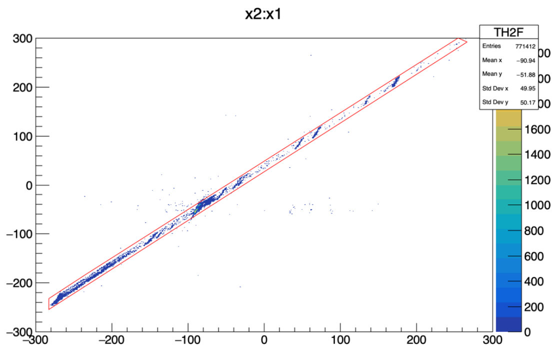
You can see that while the xavg plot only gated on $\alpha$s hasn't changed much, but our coincidence plot has become mostly empty. This is good! With a stable residual, we don't expect really any SABRE coincidences. Obviously with better gates and windows and shifts this could be reduced even further, but for the purposes of this example it demonstrates the goal and functionality of the event builder. Remember, the fast sorting is where each individual experiment will require unique conditions. For this data the anode/scintillator relationship provided a good definition of a focal plane event, and the scintillator/SABRE



(a) dE vs. xavg



(b) E vs. xavg



(c) x1 vs. x2

Figure 3: Three example histograms where a) and b) are gated on $\alpha$s and c) is gated on correlation

relationship provided a good definition of a SABRE event. This will not be the case for every experiment.

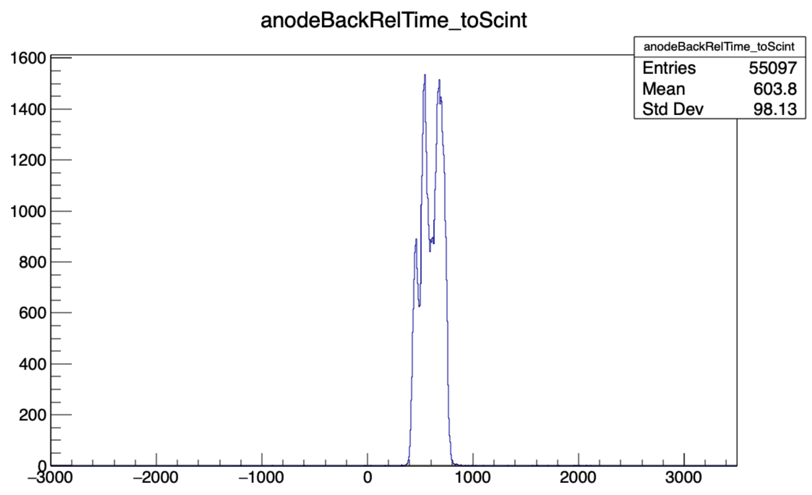
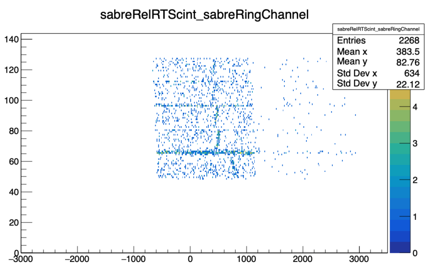
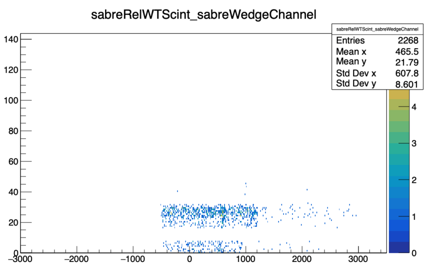


Figure 4: Plot of back anode timestamp minus left scintillator timestamp in nanoseconds



(a) SABRE rings



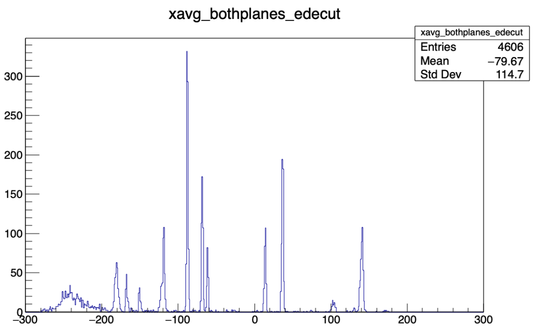
(b) SABRE wedges

Figure 5: SABRE timestamps minus left scintillator timestamps; y-axis is global channel number ((Board)16 + Channel)

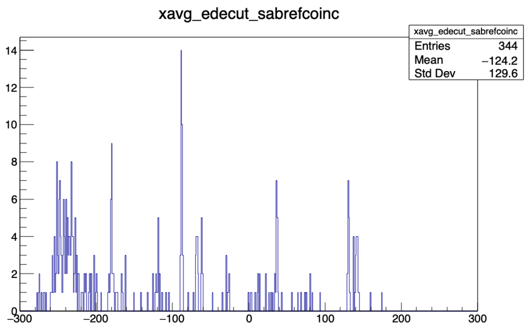
## 6 Closing remarks

This code is still under constant development, and any feedback/changes/bugs can be reported to the github remote repository where these will be taken into account. Additionally, if you have quesitons, feel free to email me at gmccann@fsu.edu.

Good luck!



(a) xavg gated on $\alpha$s



(b) xavg gated on $\alpha$s and SABRE coincidences

Figure 6: xavg plots with no shifts and wide open windows

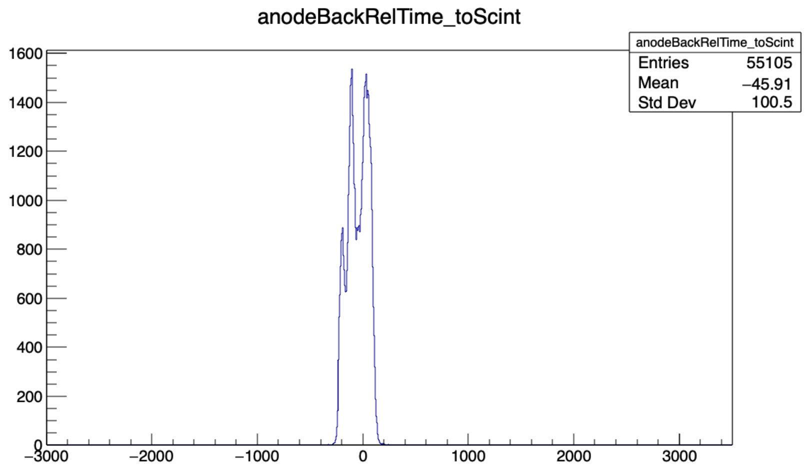


Figure 7: Anode time relative to scintillator time with a shifted scintillator

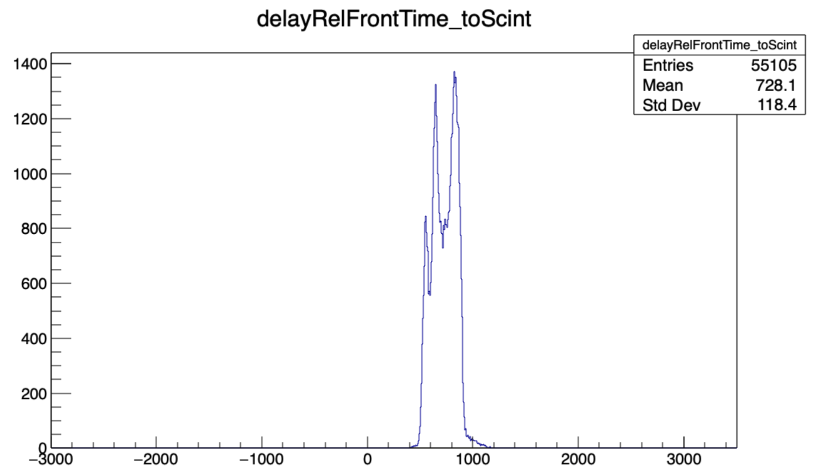
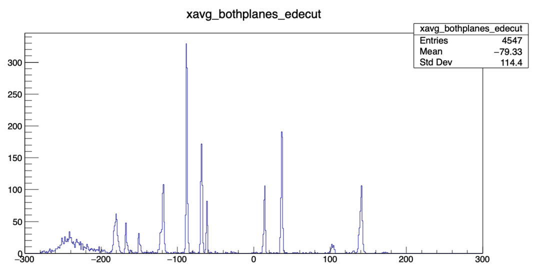
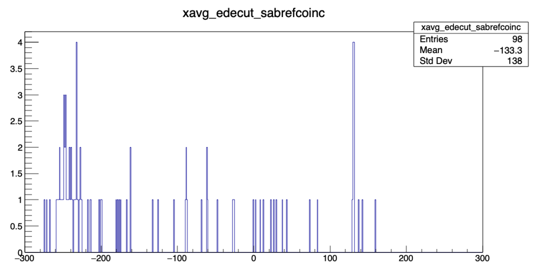


Figure 8: Maximum front delay timestamp minus scintillator timestamp



(a) xavg gated on $\alpha$s



(b) xavg gated on $\alpha$s and SABRE coincidences

Figure 9: Histograms of xavg with fast coincidence and good windows