Fall 2017 - Research Report

Jordan Stomps

February 16th, 2018

Summary

This report is designed to give an overview towards the efforts to construct an Online Data Acquisition system for the Proton Detector built by Dr. Chris Wrede's research team and associates. The research group is developing a detector that measures beta decay from proton-rich nuclides. This detector and its corresponding experimental results, will be applied to the field of astrophysics, specifically to the rates of nuclear reactions occurring during explosions on the surfaces of accreting white dwarf and neutron stars in binary systems. This gas-filled detector is mainly constructed but its subsidiary components are being optimized and developed. In the purview of this report, one system that needs to be developed is the online data acquisition system. Online in accelerator physics means while the accelerator beam is active. Currently, the infrastructure of the proton detector allows it to collect data, and then once the experiment is complete, the data can be analyzed. The research group needs a system that can actively monitor and sort the data as it is collected for the commissioning experiment of the detector in April 2018. The online system that will be utilized is a lab supported program called SpecTcl. This report should give an adequate summation of how the experiment-specific SpecTcl being developed for the proton detector operates as of December 2017.

1 How SpecTcl Interacts with DDAS

For reference, DDAS means Digital Data Acquisition System. Effectively, this is what collects data for the proton detector and sends it (using Readout) to a place that can be analyzed, like SpecTcl. A channel in the DDAS system is basically a channel for data to flow from collection and out to analysis. For example, if the proton detector has 13 channels, that means there are 13 points of data collection that are being processed and sent from Readout to SpecTcl. Readout is the program that takes events that are recognized by data collection and process them into data that can be sent to analysis software. Figure 1.1 describes what happens when data is sent to Readout.

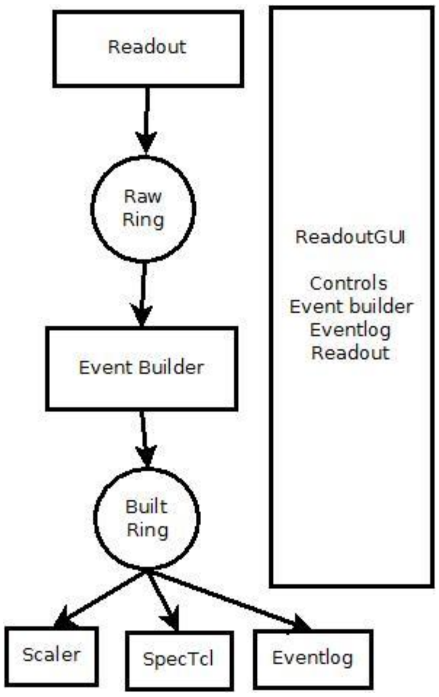


Figure 1.1 – Found at NSCL DDAS Docs (https://goo.gl/X32zXL)

As readout recognizes these data points, it packages it in an event that can then be sent to SpecTcl. If we could zoom into the bottom square titled *SpecTcl*, we would be able to break it down into Figure 1.2.

When SpecTcl is sent an event, it unpacks that event using *Event Processors* (which are each portion of code described below) and using the parameters described in initialization, it visualizes the data on a histogram.

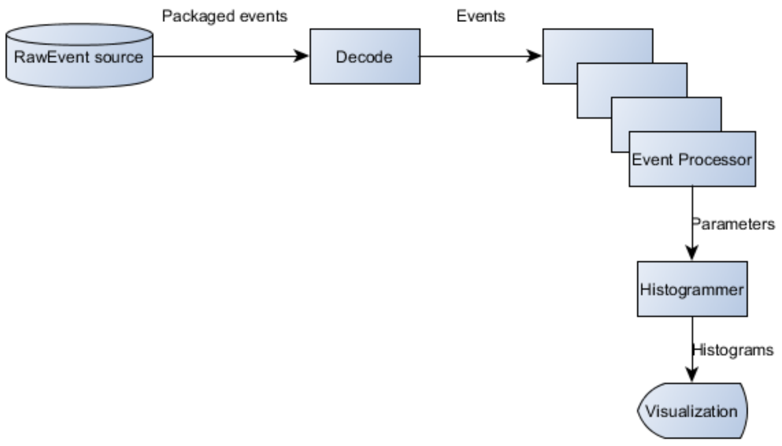


Figure 1.2 – Found in NSCL SpecTcl Guide (https://goo.gl/cgTM9X)

Both diagrams come from the user guides and resources for SpecTcl and DDAS found at http://docs.nscl.msu.edu/daq/newsite.

2 File Location

As of this writing, there are two SpecTcl builds being used. The first was pre-built and distributed amongst NSCL/FRIB. That is, the framework is readily available on the NSCL/FRIB servers and can be tailored to meet the specific needs of a project or experiment. Ideally, this is the build that the research group would like to use as it already has multiple prebuilt objects that can be used online. Currently, this code does not operate with the proton detector but work is being done to integrate it. This build will commonly be called Brent's SpecTcl in this report. The other build is one being built using the tutorials provided in SpecTcl Documentation online. The benefit of this build is that it is up-to-date and currently works using offline detector data (and is being tested to see if it works online with the detector). However, this build is very basic and does not have all the implementations needed for the experiment as of this writing. This build will commonly be referred to as SpecTcl in this report.

Each build has a location in the file system for the proton detector (this report assumes the reader is navigating the file system on a Linux computer). To find Brent's SpecTcl, open a terminal and type `cd /user/protondetector/readout/test2/spectcl-online/spectcl-online`. All necessary files for the build will be in this folder. To launch Brents SpecTcl build, simply type `./SpecTcl` in this folder. Assuming no errors occur, this will compile the source code and launch the GUI for SpecTcl. To find the source code discussed in this report, go to the directory above and type `cd src`. This folder contains all the code files that can be edited to tailor the SpecTcl build (more on this later). Currently, the objects used in Brent's SpecTcl are *SpecTcl_ddas*, *Unpacker_ddas*, *Parameters-ddas*, *Variables-ddas*, *Calibrator_ddas*, and *Threshold_ddas* (both .cpp and .h files). If at any point this SpecTcl needs to be updated or recompiled, simply go to the location of the source code and enter the command make. Assuming no errors occur, this will recompile the source code and build SpecTcl, to which it can then be launched again.

To find the folder for the standard SpecTcl build, type the command `cd /user/protondetector/readout/test2/Jordan_SpecTcl_Dev/current/Skel`. This contains both the location to launch SpecTcl and all its source code. Currently, all code files in this location are being used; *MyParameters*, *MyParameterMapper*, and *MySpecTclApp* (both .cpp and .h files). To launch SpecTcl, simply go to this directory and type `./SpecTcl`. If at any point this SpecTcl needs to be updated or recompiled, simply go to the location of the source code and enter the command make. Assuming no errors occur, this will recompile the source code and build SpecTcl, to which it can then be launched again.

For new users to editing code, there are two simple ways to edit source code files for either built:

- Type the command `geany` and then open the file in the program that is opened.
- Type the command `gedit <file-name>` where `<file-name>` is the full name (including extension) of the code to be edited.

It will often be necessary to test SpecTcl using offline data. It will be explained later in the report how to launch these data files in Spectcl, but the file location is: `/user/protondetector/ readout/test2/data/runs/complete` (just use the cd command again). Traditionally run-212-00.evt has been used to test SpecTcl (as in Figure 4.2).

3 How to Read SpecTcl Code

Because I did not build Brent's SpecTcl source code, I will not comment on what each code file means, but I will give an overview as to the source code of the standard SpecTcl build. This build was created using the tutorial Analyzing DDAS Data in SpecTcl Tutorial found here: http://docs.nscl.msu.edu/daq/newsite/ddas-1.1/ddas_spectcl.html. As of this writing, the build has not strayed far from the code displayed in the tutorial. For that reason, while it may change in the future, this report will only give supplemental feedback to the comments in the tutorial. Ultimately, if this build is used for the experiment, the code will certainly stray from the tutorial as cuts and other parameters will need to be implemented.

Simply put, there is only a small distinction between .cpp and .h files. .h files are for declarations, whereas .cpp files are for definitions. Because of that, .h files have a #include line in .cpp files. This is just to help organize code:

3.1 MyParameters.h

This creates a tree structure for each parameter (or channel) of data. Basically, tree structure is a way of grouping like-parameters together. Currently, the code creates 48 parameters that are associated with potentially 48 channels of raw data. Note that for our experiment, there are not 48 channels worth of data. Instead, we have 13 channels for the proton detector and (eventually) 16 channels for SeGA. Fortunately, SpecTcl does not care how many channels are initialized as DDAS will assign a channel to each event hit. Thus, within SpecTcl, only the channels that are desired for viewing need to be selected and the rest can be ignored.

3.2 MyParameters.cpp

This initializes each of the 48 parameters. The first class creates the framework for each channel. It creates an energy and timestamp for each channel. The original code prepares spectra with 4095 bins. For our experiment, there are 65536 bits worth of data being collected. However, if we binned this, it would produce large amounts of data that would not be feasible for SpecTcl. 4095 may be a reasonable range for our measurements, but fortunately these parameters can be changed during SpecTcl initialization. The timestamp is registered in nanoseconds and has a similar structure to how energy is initialized. This can easily be changed to fit our experiment. Remember, these settings are simply to help visualize the data in a spectrum, they do not affect the data that is being collected from DDAS. With that in mind, these settings will need to be tailored for our specific experiment. The second class runs through a loop to create an energy and timestamp parameter for each channel and it also creates a multiplicity spectra to see which channels of the detector collected data.

3.3 MyParameterMapper.h

This code uses a specific object that is already built to communicate with DDAS so that event hits can be unpacked and prepared for SpecTcl. This object is a ddaschannel object that knows the format in which DDAS Hits can be unpacked. In the code, this separate class is referred to as DAQ::DDAS::DDASHitUnpacker. Information on this class can be found at its reference page located here: http://docs.nscl.msu.edu/daq/newsite/ddas-1.1/classDAQ_1_1DDAS_ 1_1DDASHitUnpacker.html. The benefit of using this class is that it knows how to communicate with DDAS so we dont have to create an object or class that manually unpacks hits (an example of how a manual unpacker is coded can be found in Brent's SpecTcl: `Unpacker_ddas.cpp` and `Unpacker_ddas.h`). The reason Brent's SpecTcl was unsuccessful at unpacking our event files was because it unpacked data in a way that was incompatible with the DDAS data we collected. For this reason, we should continue to use `DAQ::DDAS::DDASHitUnpacker` while it works with our detector and DAQ system.

3.4 MyParameterMapper.cpp

This file takes the unpacked event hits from the above code and assigns them to the proper channel index based on the information that was unpacked. As of this writing, this code works properly with the data we are collecting, and thus has not been changed extensively from the coding tutorial.

3.5 MySpecTclApp.h and MySpecTclApp.cpp

These files are what initializes SpecTcl with all the additional code compiled. This can mostly be left untouched but it is important to note an addition to `MySpecTclApp.cpp`. In this section, we ensure that `DDASBuiltUnpacker.h` is included to help unpack the DDAS data. The parameter name (currently raw for each channel because it consists of the raw data) is initialized in this section.

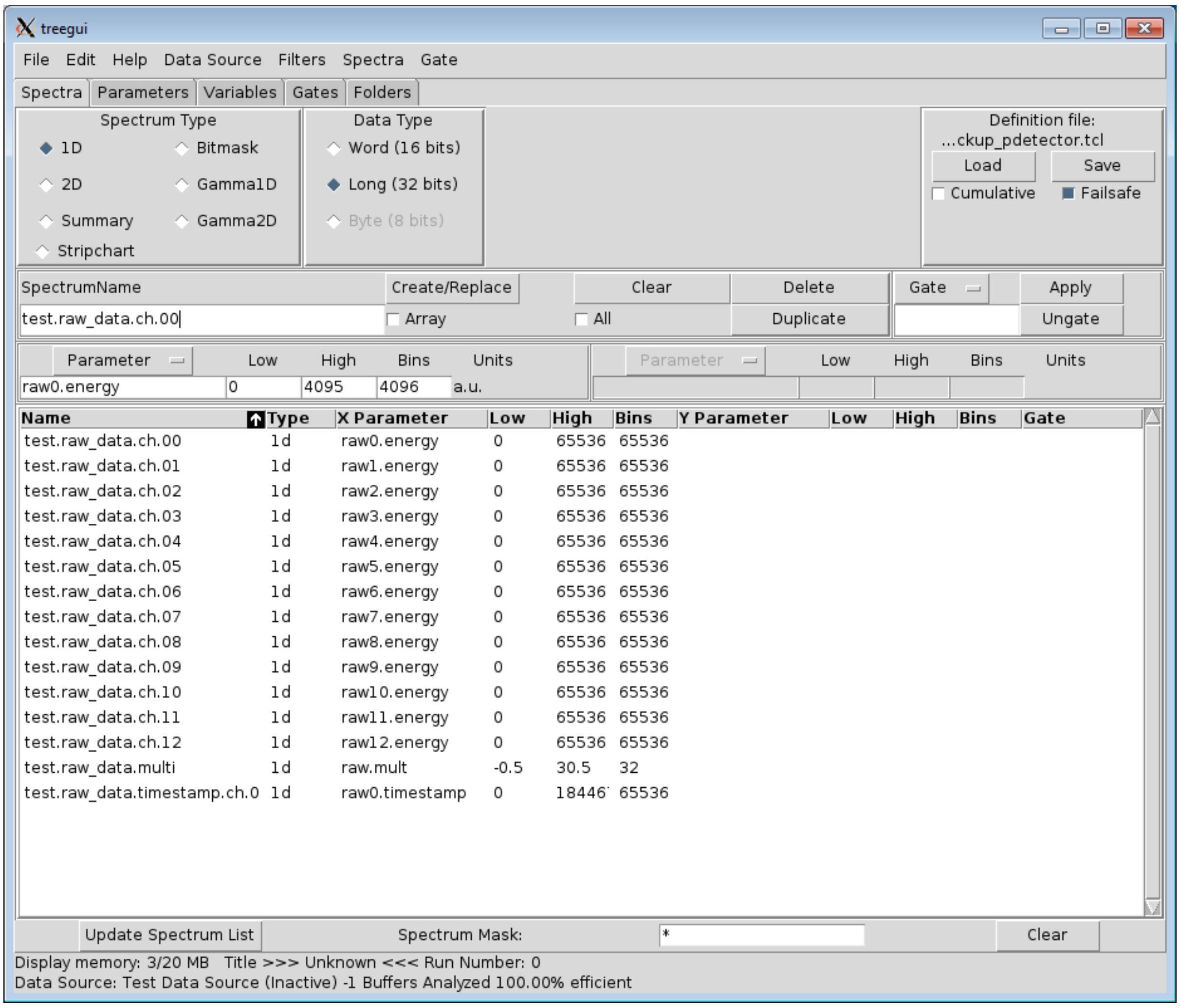
3.6 Makefile

This file contains the code that compiles a SpecTcl build that can be used. Of notable importance, each code file must be added as an object variable in the `OBJECTS=` line. Additionally, the `USERCXXFLAGS=` and `USERLDFLAGS=` lines must specify that DDAS code must be compiled with SpecTcl.

Once each code file has been written, SpecTcl can be compiled by typing `make` in the command line where the Makefile is located. When this has been accomplished, and assuming no errors are encountered, SpecTcl can then be run with the command given above. Compiling only needs to occur after changes to the code have been made, not before every time SpecTcl is opened.

4 How to Operate SpecTcl

The goal of this section is to briefly explain how the SpecTcl GUI is used to produce spectra and connect to data sources. Once SpecTcl has been started, the treegui will be initialized. To add parameters or channels. Go to the `Spectra` tab and type in a name for the channel under `SpectrumName`. I usually choose something like *test.raw_data.ch.01* for channel 1 event hits as seen in Figure 4.1. For the actual experiment, or when collecting data, it is more appropriate to type *proton_detector* instead of *test* for example. Then select `Parameter` below this and select the appropriate parameter to be created. At this point, the limits or bins can be changed or the default can be kept. Once these things have been filled out, `Create/Replace` can be selected to add this parameter to those that are saved. To avoid repeating the tedious task of creating every channel/parameter, once this task has been satisfactorily completed, the `Save` button can be selected in the top-right and a definition file can be created and loaded for later use.


treegui
File Edit Help Data Source Filters Spectra Gate
Spectra Parameters Variables Gates Folders
Load Save
Create/Replace Clear Delete Apply
Duplicate Ungate
Update Spectrum List Clear

Spectrum Type: [✓ 1D] Bitmask 2D Gamma1D Summary Gamma2D Stripchart

Data Type: Word (16 bits) [✓ Long (32 bits)] Byte (8 bits)

Definition file: ...ckup_pdetector.tcl

Cumulative [✓ Failsafe]

SpectrumName: test.raw_data.ch.00 Array All Gate

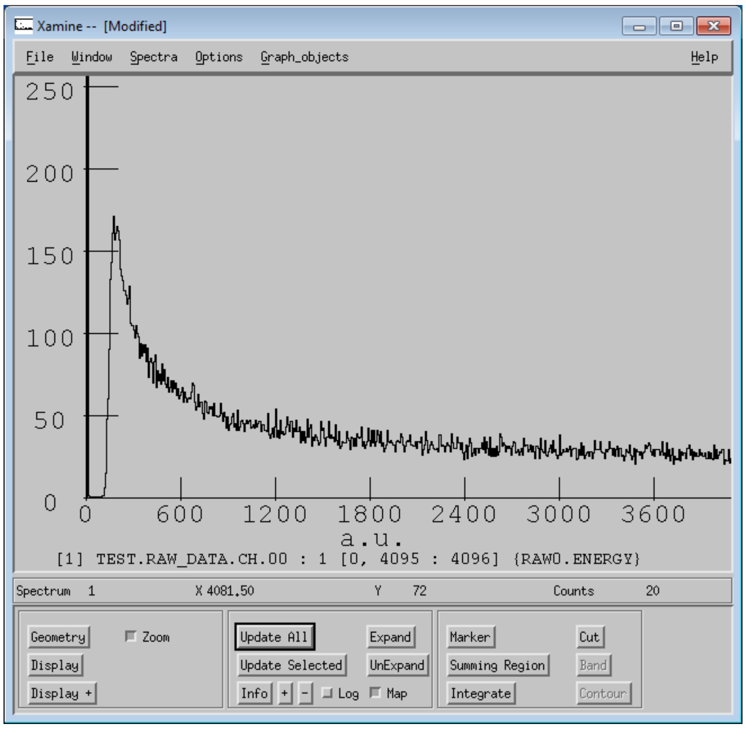
Parameter	Low	High	Bins	Units
raw0.energy	0	4095	4096	a.u.

Name	Type	X Parameter	Low	High	Bins	Y Parameter	Low	High	Bins	Gate
test.raw_data.ch.00	1d	raw0.energy	0	65536	65536					
test.raw_data.ch.01	1d	raw1.energy	0	65536	65536					
test.raw_data.ch.02	1d	raw2.energy	0	65536	65536					
test.raw_data.ch.03	1d	raw3.energy	0	65536	65536					
test.raw_data.ch.04	1d	raw4.energy	0	65536	65536					
test.raw_data.ch.05	1d	raw5.energy	0	65536	65536					
test.raw_data.ch.06	1d	raw6.energy	0	65536	65536					
test.raw_data.ch.07	1d	raw7.energy	0	65536	65536					
test.raw_data.ch.08	1d	raw8.energy	0	65536	65536					
test.raw_data.ch.09	1d	raw9.energy	0	65536	65536					
test.raw_data.ch.10	1d	raw10.energy	0	65536	65536					
test.raw_data.ch.11	1d	raw11.energy	0	65536	65536					
test.raw_data.ch.12	1d	raw12.energy	0	65536	65536					
test.raw_data.multi	1d	raw.mult	-0.5	30.5	32					
test.raw_data.timestamp.ch.0	1d	raw0.timestamp	0	18446	65536					

Spectrum Mask: *

Display memory: 3/20 MB Title >>> Unknown <<< Run Number: 0
Data Source: Test Data Source (Inactive) -1 Buffers Analyzed 100.00% efficient

Figure 4.1



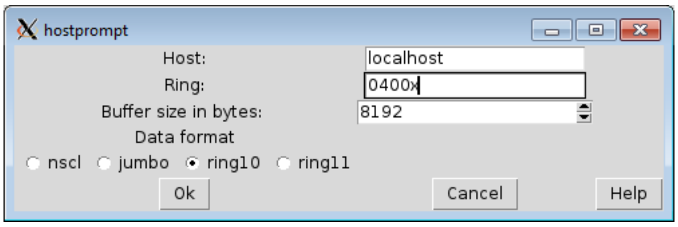
[1] TEST.RAW_DATA.CH.00 : 1 [0, 4095 : 4096] {RAW0.ENERGY}

Spectrum 1 X 4081.50 Y 72 Counts 20

Figure 4.2

To connect to a data source, select `Data Source` in the top-left menu bar. If you want to connect an event file, select `File` and a new GUI will be selected. From here, you can use the left pane to navigate (use the .. to go back a directory) and the right pane to select event files available in the specified directory. Once the file has been selected, choose the appropriate ring (either Ring10 or Ring11) and select `Ok`. Once each buffer has been analyzed, the data will be compiled in SpecTcl and the Xamine GUI can be selected. Use `Display` in the bottom-left to select which channel should be displayed in the viewpane. You can also press `Update All` if any changes have been made to the data or spectrum. If these steps are conducted correctly, an example of what Xamine should look like is displayed in Figure 4.2.

To attach an online data source, select `Data Source` in the top-left menu bar and choose `Online..`. Then, fill out Host and Ring using the appropriate Readout settings. While this has not been tested, the correct values should be `localhost` for Host (assuming SpecTcl is running on the same computer that DDAS is) and `0400x` for Ring, as seen in Figure 4.3. Also make sure to select the correct data format (either Ring10 or Ring11) and select Ok. Then, just as with the event file, Xamine can be used to view spectra as the data is collected from DDAS.



Host: localhost
Ring: 0400x
Buffer size in bytes: 8192
Data format
nscl jumbo [✓ ring10] ring11

Figure 4.3

5 Looking Forward

This is currently the extent to which SpecTcl has been created and used. That said, I should highlight a few things that will need to be implemented and tailored to SpecTcl before it is truly ready to be used experimentally:

- Channels and parameters for SeGa (or any other additional data sources) will need to be added to the parameters available in SpecTcl for the actual experiment.
- Cuts/Slices used to analyze data from the detector will need to be implemented. This should not be hard as Moshe Friedman already has these created in Root code. Thus, the Root code will simply need to be translated and added to SpecTcl in a way that it can understand and compile.
- Most importantly, online data collection needs to be tested to ensure it works with our detector. While most of the information needed to collect data online with SpecTcl is known, it has not actually been tested for issues while communicating with DDAS.