2015

NSCLDAQ 11.0 User's Guide

Last modified: 1/13/2015

TOMPKINS, JEROMY

1 CONTENTS

2 INTRODUCTION TO NSCLDAQ 11.0 AND THIS DOCUMENT

The motivation for the next major revision of NSCLDAQ, 11.0, is event building. It has become more common for experiments to couple multiple data acquisition systems together into a unified entity and then attempt to correlate the data from these systems in the online setting. Though there was some minimal support for this in NSCLDAQ 10.2, it was limited and hard to extend. NSCLDAQ 11.0 provides a much more flexible system for controlling independent subsystems of the DAQ, a framework for processing data online, a data format that intrinsically contains a timestamp and origination identifier, better support for merged data streams, and improvements to the event builder.

The flexibility for controlling the different subsystems is provided by the **brand new ReadoutGUI**. It looks similar to the 10.2 version, but it is rewritten to be independent of any Readout program. Whereas in 10.2, the ReadoutGUI was associated with a specific Readout program, the 11.0 ReadoutGUI begins without knowledge of any Readout programs. The user can then register an arbitrary number of Readout programs to it to control. Setting up a ReadoutGUI to control a multiple Readout programs is just about as easy as setting it up to control a single Readout program. Other things can be registered for control besides a Readout program. The most useful of these is probably the registration of a separate ReadoutGUI for remote control, because it makes modularization of the DAQ possible. The new ReadoutGUI is a significant enhancement over the 10.2 version.

For operating on the data stream in the online setting (or offline), NSCLDAQ 11.0 introduces the **filter framework**. This is a framework that simplifies the task of creating a program to read data from a data source, operate on it, and then output the result to a data sink. These sources and sinks can be either files, stdin or stdout, or ring buffers. So far the uses for it have been data integrity checking, data formatting, filtering out unwanted data, and analysis. It has become an integral tool in NSCLDAQ.

The advent of the event builder in version 10 of NSCLDAQ brought merged data streams into existence. In a merged stream there are duplicate types of data that were contributed by different Readout programs. Sorting out the point of origin of each item of data requires that an origination identifier be associated with each data object. Furthermore, to reliably merge the independent streams together in a time-ordered fashion, the data items must contain a key by which to order them by. NSCLDAQ solves these problems by inserting into the data format a concept called a **body header**. This is an optional piece of information and is not always present. When present it contains the origination identifier and the ordering key, which are referred to as the source id and the timestamp, respectively. The introduction of the body header is critical to understanding the data downstream of the event builder and serves as a major enhancement from the 10.2 data format.

Besides identifying and understanding the data within a merged data stream, there were some issues in NSCLDAQ 10.2 programs concerning when to consider a run complete. In pre-event-builder times there was only ever one begin run and one end run item in a data stream. In a merged stream, there is as many of these as there were data sources upstream. In 10.2, the program responsible for recording data to disk, eventlog, would exit after the first end run item it saw. If the stream of data it was recording was a merged stream, the subsequent end runs would not be saved into the data file. NSCLDAQ 11.0

improved the logic for detecting the end of a run to solve this problem. The user provides to it the number of end run items it should find and it will only exit after that many have been observed.

Finally, it would be wrong to say that the goal of NSCLDAQ 11.0 was to improve support for event building and then not mention any **improvements to the event builder** itself. The new major version brings with it a host of improved diagnostics. Suspiciously bad occurrences such as duplicate timestamps, out-of-order fragments, and late fragments are now all reported to the user when identified. Other diagnostics like input and output rate for each source id have also been made available as well to help evaluate the health of the entire system. Finally, flow control has been introduced to the event builder to prevent it from continually accepting data from clients while it cannot output data itself. This prevents it from running out of resources and also allows the upstream data providers to be notified that the data stream has backed up. Together these make the event builder a much more usable and informative tool.

Because NSCLDAQ 11.0 is a major revision, existing 10.2 systems will need to be updated. Compatibility was allowed to be broken in this revision, but it was maintained whenever it made sense. The remaining two sections of this document intend to provide the user with the tools and information needed to make the transition from 10.2 to 11.0 as smoothly as possible. The first section focuses on some clear things that will be needed to transition a 10.2 system to an 11.0 system. The second is for the user who is starting from scratch and needs to set up a fully functional 11.0 system. There are enough changes in some areas of NSCLDAQ 11.0 that sometimes it is easier for the user to simply start from scratch and that approach is sometimes taken in the tutorial.

3 MIGRATING FROM 10.2 TO 11.0

As described above, there were major changes introduced in 11.0 that are likely to break existing setups that worked in 10.2. It is impossible to identify how to update every little change in a single document and this does not seek to do so. The goal of this section is to identify some things that clearly must be addressed by the user as well as some guidance for how to do so.

3.1 DATA FORMAT

The most major change between 10.2 and 11.0 was the data format. In 10.2, the data format was a structure that had the form:

Table 1 Structure of an NSCLDAQ 10.2 RingItem

	Description	Size (bytes)	Offset (bytes)
Header	Inclusive size	4	0
	Type	4	4
Body	Data…	Greater than or equal to 0	8

In the 11.0 data format, the idea of a “body header” has been introduced. The body header is optional and is inserted in between the header and the body. It contains the timestamp, source id, and barrier type, each of which are useful for event building. Together this is an extra 20 bytes of data. To identify

whether or not the body header is present, there is also a body header size always following the header. If this size if 0, there is no body header and the next 32-bit integer is the start of the body data. However, if the size is 20, there is a body header presenter. See the difference in the following two tables.

Table 2 Structure of an NSCLDAQ 11.0 RingItem without a body header.

	Description	Size (bytes)	Offset (bytes)
Header	Inclusive size	4	0
	Type	4	4
Body Header	Size = 0	4	8
Body	Data…	Greater than or equal to 0	12

Table 3 Structure of an NSCLDAQ 11.0 RingItem with a body header

	Description	Size (bytes)	Offset (bytes)
Header	Inclusive size	4	0
	Type	4	4
Body Header	Size = 20	4	8
	Timestamp	8	12
	Source id	4	20
	Barrier type	4	24
Body	Data…	Greater than or equal to 0	28

Adjusting user code to handle the new data format is really as simple as checking for the presence of the body header and acting accordingly.

The other bits to recognize about the data format are the change in some ring item types. In NSCLDAQ 10.2, the ring item types were BEGIN_RUN, END_RUN, PAUSE_RUN, RESUME_RUN, PACKET_TYPES, MONITORED_VARIABLES, INCREMENTAL_SCALERS, TIMESTAMPED_NONINCR_SCALERS, PHYSICS_EVENT, PHYSICS_EVENT_COUNT, EVB_FRAGMENT, and EVB_UNKNOWN_PAYLOAD. In NSCLDAQ 11.0, the INCREMENTAL_SCALERS and TIMESTAMPED_NONINCR_SCALERS are replaced with a single type called PERIODIC_SCALERS and some new item types are added. Those are ABNORMAL_ENDRUN, RING_FORMAT, and EVB_GLOM_INFO.

3.2 ENVIRONMENT VARIABLES

Be sure to update the DAQROOT, DAQLIB, and DAQBIN environment variables to refer to a proper 11.0 distribution. Setting these is as easy as executing the following two commands at the command line:

```
% unset DAQROOT
% source /usr/opt/nscldaq/11.0/daqsetup.bash
```

It would actually be worthwhile adding these two lines to your .bashrc file or modifying them if they already exist.

Often scripts and Makefiles have the paths to NSCLDAQ executables or directories hardcoded into them. It is a good time to locate those scripts and update them too. Rather than updating the hardcoded path,

from /usr/opt/nscldaq/10.2-108 to /usr/opt/nscldaq/11.0, it would be preferable to make the path dependent on the DAQROOT, DAQBIN, or DAQLIB environment variables. Adopting this approach will help avoid future surprises caused by unknowingly running some parts of the system as 10.2 and other parts as 11.0.

3.3 VMUSBREADOUT AND CCUSBREADOUT

There are no changes to the VMUSBReadout and CCUSBReadout programs that will break compatibility with version 10.2 because only new functionality has been introduced. It is strongly encouraged that the user choose to provide the –timestamplib and –sourceid command line switches. By doing so, all ring items produced by the Readout program will have body headers assigned to them.

3.3.1 Creating a timestamp extractor for VMUSBReadout or CCUSBReadout.

For the xxUSBReadout programs, there are two timestamp extractors that can be defined for extracting a timestamp from the data, one is for the physics event data and the other for the scaler data. The goal of each of these functions is to produce a 64-bit integer computed from the data (or something else). It is okay to only implement on of the functions but is not recommended. The consequence for missing on of the functions is that the body header will not be created for that associated type of item. Sample source implementations of these two methods can be found in Section 5.1. Take special note of the getScalerTimestamp(void*) item in it because it is implemented to cause the scaler items to pass through the event builder in the order they were received.

The source code must be compiled into a dynamically-linked shared object in which the symbols are not mangled (see sample file for how to do this). Assuming the file is named rdotstamp.cpp, you can compile it into the proper format via:

```
% g++ -shared –o librdotstamp.so –I$DAQROOT/include rdotstamp.cpp
```

You would then pass "/path/to/librdotstamp.so" with the –timestamplib command line switch.

3.4 SBS READOUT FRAMEWORK

As in the VMUSBReadout and CCUSBReadout, little has changed that would break existing code. The major actions the user should consider taking are adding a timestamp and source id to the body of their data just like in the xxUSBReadout programs. Unlike the VMUSB/CCUSBReadout programs, this is done via API calls within the source code of their application. To set a timestamp, the user should call the CEventSegment::setTimestamp(uint64_t) method within the implementation of their CEventSegment::read(void*,size_t) method. To set the source id, call the CExperiment::setDefaultSourceId(uint32_t) method in your Skeleton::SetupReadout(CExperiment*) method.

3.5 SCALERDISPLAY

In NSCLDAQ 10.2, there was ambiguity concerning the channel index if it was attached to a ring containing merged data streams because there was no source id provided to the channel command. For this reason, if two data streams each contributed 32 scaler channels and then were merged together, the values in the ScalerDisplay for each channel would alternate back and forth between the values of

the two data streams. In NSCLDAQ 11.0, this is fixed by providing the source id with the index when calling the channel command. The new signature of the channel command is:

channel [?options?] *name index*[?.sourceid?]

In the situation just described, if the two streams were identified as 0 and 1 and the first channel of each was a 10 Hz clock and a live trigger count, one might set this up in their configuration file with the following two lines.

```
channel  10HzClock    0.0
channel  LiveTrigger  0.1
```

Remember that just because the system is 11.0, it doesn't mean that the data has a body header with it identifying the source id. What is demonstrated above only should be done if the scaler items have a body header. Otherwise, drop the decimal point and subsequent source id because they have no meaning.

3.6 READOUTGUI

The ReadoutGUI changed a lot between 10.2 and 11.0. It is such a change that it is worth working through Section 4.3.

With that said, the ReadoutCallouts.tcl file operates in much the same way as it did. There are still the OnBegin, OnEnd, OnPause, OnResume, and OnStart callbacks and an addition OnFail proc, which is called whenever a state transition fails, has been added. If an existing ReadoutCallouts.tcl file made previous calls to the ReadougGUIPanel namespace, the user will have to update their script. For the most part, the only thing that has changed is ReadougGUIPanel is replaced with ReadoutGUIPanel. See Table 4 for a mapping of commands from the 10.2 format to the 11.0 format.

Table 4 Mapping of TCL procs in 10.2 to 11.0

NSCLDAQ 10.2	NSCLDAQ 11.0
ReadougGUIPanel::getHost	No corollary
ReadougGUIPanel::getPath	No corollary
ReadougGUIPanel::setRun	ReadoutGUIPanel::setRun
ReadougGUIPanel::getRunNumber	ReadoutGUIPanel::getRun
ReadougGUIPanel::incrRun	ReadoutGUIPanel::incrRun
ReadougGUIPanel::setTitle	ReadoutGUIPanel::setTitle
ReadougGUIPanel::getTitle	ReadoutGUIPanel::getTitle
ReadougGUIPanel::recordData	ReadoutGUIPanel::recordData
ReadougGUIPanel::isTimed	ReadoutGUIPanel::isTimed
ReadougGUIPanel::setTimed	ReadoutGUIPanel::setTimed
ReadougGUIPanel::getRequestedRunTime	ReadoutGUIPanel::getRequestedRunTime
ReadougGUIPanel::setRequestedRunTime	ReadoutGUIPanel::setRequestedRunTime
ReadougGUIPanel::recordOn	ReadoutGUIPanel::recordOn
ReadougGUIPanel::recordOff	ReadoutGUIPanel::recordOff

3.7 EVENT BUILDER

The interface for using the event builder is left mostly the same, but the user will at least need to update his/her timestamp extractors to reflect the new data format. All else does not *need* to change but the recommended method for enabling event builder support in the ReadoutCallouts.tcl file has changed. As long as no failures happen, the old way of enabling the event builder will still work fine. However, it is possible that failed state transitions could induce a situation where the ReadoutGUI locks up indefinitely. Please seriously consider using the new scheme for enabling the event builder as described in Section 4.4.

4 SETTING UP AN 11.0 SYSTEM FROM SCRATCH

At some point in the life of any experiment at the NSCL, someone is going to have to set up the software to control data taking. This may seem like a daunting task at first glance, but it should not be feared. In reality, the procedure is only as complicated as the experiment demands. For an experiment that has only a single VME crate to read out, the set up procedure is actually quite simple. For an experiment that seeks to read out two or more VME crates and merge the data using the NSCLDAQ event builder, it is a bit more involved but is straightforward to accomplish. The purpose of this document is to provide some basic procedures that will help an experimenter build a fully-functional system from scratch. The experimenter is urged to set up the full DAQ that will run during their experiment as early as possible so that the entire DAQ can be tested sufficiently prior to beam time.

4.1 INTENDED AUDIENCE:

Every experimenter. The goal is that very minimal knowledge should be needed to set up an experiment. In the simple case of reading out a single VME crate, there is no need to write any code. If the event builder will be used, some basic knowledge of C/C++ is needed to define how a timestamp is to be extracted from the data.

4.2 ASSUMPTIONS:

There are a few assumptions that are being made in the following tutorial.

1. The electronics for the DAQ are already setup and are connected with the host.
2. The reader knows basic usage of the Linux command line and understands how to standard programs like ssh.
3. NSCLDAQ has already been built and installed. Furthermore, the user knows the path to the version of NSCLDAQ they intend to run. (On NSCL computers, you can find all deployed versions in the /usr/opt/nscldaq directory).

4.3 TUTORIAL FOR COMMON SET UP TASKS:

The approach taken by this tutorial will be to build up a system piece by piece. It will begin by establishing that subsystems of the DAQ are capable of operating independently using the command line and then move toward running multiple subsystems through a single, unified GUI.

4.3.1 Establish the ability to run each DAQ subsystem at the command line

My recommendation is that no matter the setup, whether a single VME crate or multiple VME crates, the user should be able to run each Readout program independently from the command line before proceeding further. If you cannot run your Readout program from a command line, it won't work in the future when run through a GUI. Multi-subsystem DAQs that share a trigger can be rewired to be in the final experimental configuration once each subsystem is understood to be capable of running on its own.

Most people do not know how to run a Readout program at the command line. It is actually straight forward. For any NSCLDAQ readout program, a TCL interpreter prompts the user for input once started up at the command line. To begin taking data, simply type "begin". To stop data taking, type "end".

It is not considered normal behavior to experience segmentation faults, uncaught exceptions, double frees, or any other hard crash when transitioning to, from, or during data taking. **Remember that instabilities found while testing become instabilities during production beam time if not addressed early enough to fix them. When observing these sorts of behaviors, contact the appropriate channels.**

While the DAQ is running, attach the dumper program to the ringbuffer that is receiving the data to verify that the data is in fact present and that it looks reasonable. It is also useful to attach a SpecTcl as well if possible. Only once the data from the Readout program being tested is sensible and understood to be stable, should the reader move on.

Repeat this test for each of the Readout programs that will be run during the experiment.

4.3.2 Make sure your Readout programs produce data with a timestamp and source id

If you are running as single crate system with no intention of using the event builder, then feel free to move on to the next section. Anyone else would put themselves in the best possible position by providing their Readout program with these important piece of information. In an SBS Readout program, the user will have to program this capability into their event segments and skeleton file. It is a bit more involved. However, if you are using the VMUSBReadout or CCUSBReadout programs, you would accomplish this task by providing the –sourceid and –timestampextractor command-line options when launching the program. The source id should be a unique integer identifying a single Readout program. If your system is composed of four Readout programs, you should have four source ids, one for each Readout program. The argument passed to the –timestampextractor option requires more explanation. See Section 3.3.1 for a brief tutorial.

Why is it a good idea to do this? If the Readout program does not add the timestamp and source id to the data, that information will be bundled with raw data item immediately before being passed into the event builder. When the data is outputted from the event builder, the physics event data and the other types of data are treated differently. The physics events event data remains packaged as the raw data item, the timestamp, and source id. For all other types of data (begin run, end run, scaler, etc.), only the raw data is outputted; the timestamp and source id bundled with the raw data are discarded. What this means is that if you are merging multiple data streams from Readout programs lacking the timestamp and source id, it may not be possible to identify which pieces of non-physics event data came from which data stream downstream of the event builder. On the other hand, if the Readout program supplied the timestamp and source id, that information will be embedded into the raw data items in the form of a body header. Such information will not be removed because it is part of the raw data item.

4.3.3 Set up the system for passwordless login over ssh

At this point, it is assumed that the Readout programs are functional entities. The next step is the make it possible to launch the Readout program(s) and control them with the ReadoutGUI. Because most Readout programs are launched by the ReadoutGUI as ssh pipes, it is paramount that the user can login to their own account without being prompted for a password. To accomplish this, follow the instructions at https://portal.frib.msu.edu/nscloperationsdivision/it/Knowledgebase/Setting%20up%20SSH%20so%20you%20don%27t%20need%20to%20type%20a%20password.aspx. It should be easy to determine whether this has been set up properly by typing:

```
% ssh localhost
```

If properly set up, no password should have been requested. If you were prompted for a password, then something is amiss. Go back over the directions and make sure you followed them all. If you are still being prompted for a password, contact someone that can help with this sort of issue. At the NSCL, you're best solution is to email helpme@nscl.msu.edu and providing a very detailed description of the behavior and a copy of all the error messages you received.

4.3.4 Establish useful environment variables

Most people do not realize that NSCLDAQ provides a convenient script to define some useful environment variables. These are the DAQROOT, DAQLIB, and DAQBIN variables. Set these all to the desired NSCLDAQ version by executing the following at the command line:

```
% unset DAQROOT
% source /usr/opt/nscldaq/11.0-xxx/daqsetup.bash
```

To automate this every time you login, add these lines to your.bashrc script. This is actually a recommended usage pattern.

It is also useful to inspect your environment variables, specifically TCLLIBPATH, to make sure that they are not referring to conflicting NSCLDAQ versions. Unless you *need* the TCLLIBPATH to be set, it is recommended to unset it.

```
% unset TCLLIBPATH
```

Another useful thing to do is to add DAQBIN to the PATH environment variable by executing:

```
% export PATH=$PATH:$DAQBIN
```

From here on, it will be assumed that the PATH variable has been altered to include DAQBIN.

4.3.5 Create the stagearea

The "stagearea" is a hierarchy of directories with well-defined structure that ReadoutGUI uses to store data files. Typically, the toplevel directory of this is named stagearea. Because large volumes of data will be stored in this directory and only 2 GB are made available on the home directory of a standard experimental account, this directory should not actually live in the user's home directory. The canonical and recommended way of setting this up is to create a symbolic link in the user's home directory that points to the actual directory to be used for storage. For an experimental account e12003, the actual directory for storage would be at /events/e12003 and a symbolic link should be created by typing

```
% ln –s /events/e12003 $HOME/stagearea
```

4.3.6 Start the ReadoutGUI

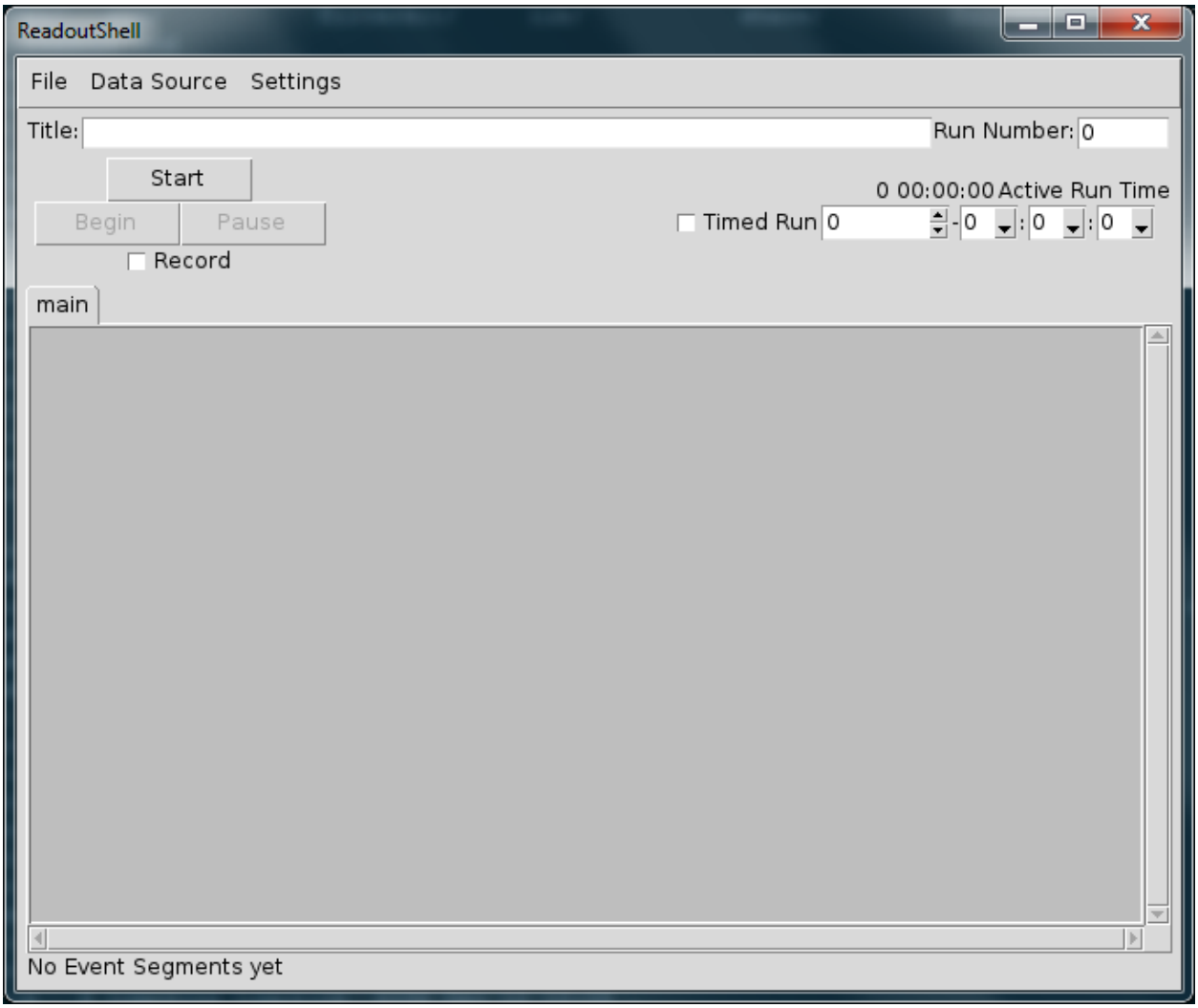
To launch the ReadoutGUI, the user should execute the following command:

```
% ReadoutShell
```

A note on hard-coding paths into scripts… A lot of people like to put this kind of code into a bash script named something like "godaq". Many times, these scripts contain hardcoded path of the executable (i.e. /usr/opt/nslcdaq/11.0-rc9/bin/ReadoutShell). Though there are good reasons to do this within a given experiment, it is dangerous if your group has a tendency of copying configuration files from old experiment accounts into new ones. Doing so can lead to version conflicts that may lead to unexpected behavior. It is preferable that if the user wants to provide explicit versioning in their scripts that they do so by sourcing the version specific daqsetup.bash script in their .bashrc and then referring to executables via the DAQBIN environment variable. By following this paradigm the godaq script would contain $DAQBIN/ReadoutShell rather than /usr/opt/nscldaq/11.0-rc9/bin/ReadoutShell. The benefit of this is that the definition of the version is centralized. You could roll forward an old account and just change the version of daqsetup.bash that is sourced in the .bashrc and it would ensure that any script referencing DAQBIN runs the intended version.

At this point, one of three things should have happened.

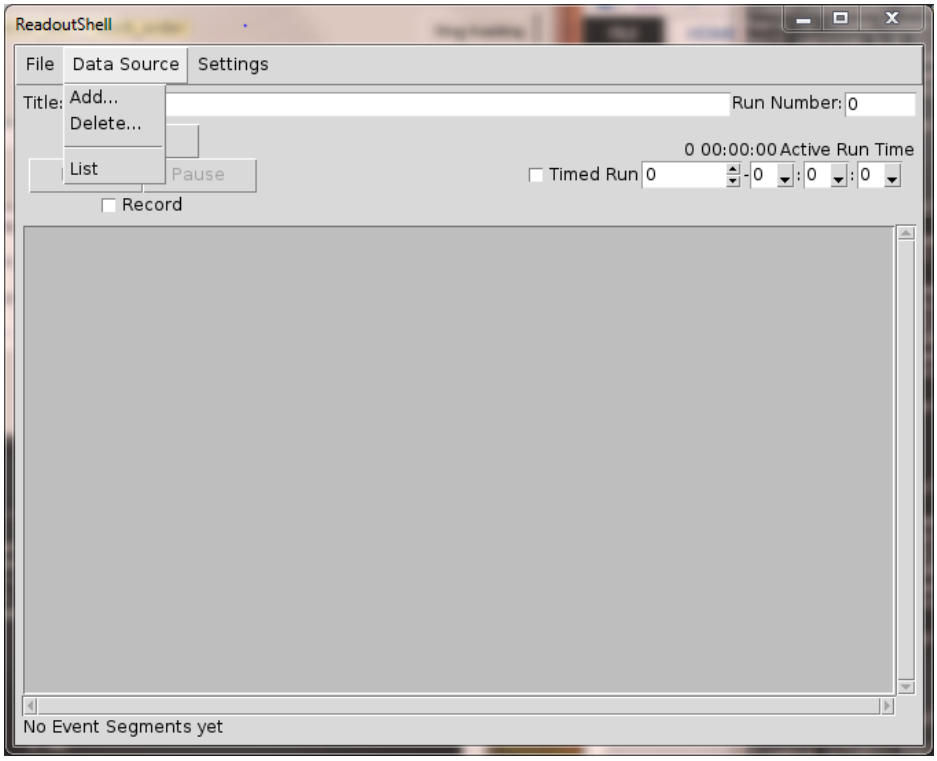
1. The ReadoutGUI became visible.
2. You were presented with a message about TCLLIBPATH_OK. (This means you did not heed the advice I already gave about unsetting the TCLLIBPATH. For this reason, I would guess you are able to assess whether or not what you added to your TCLLIBPATH is going to conflict with the TCL code of NSCLDAQ. That code is found in $DAQROOT/TclLibs and $DAQLIB.) In which case, follow the directions of the prompt and act appropriately for your situation.
3. An error occurred that prevented the ReadoutGUI from becoming visible. Here are some common reasons for this at the NSCL
    a. Being logged in over ssh without the –X or –Y flag. (This applies to PUTTY as well)
    b. XMing is not running (only applicable if using an office desktop)
    c. Something else is going on. Can you open any applications that produce a GUI? Consider contacting helpme@nscl.msu.edu to help diagnose why you are unable to view a window. If you do, please be as explicit as possible and include either a screenshot or verbatim copy of any error message.



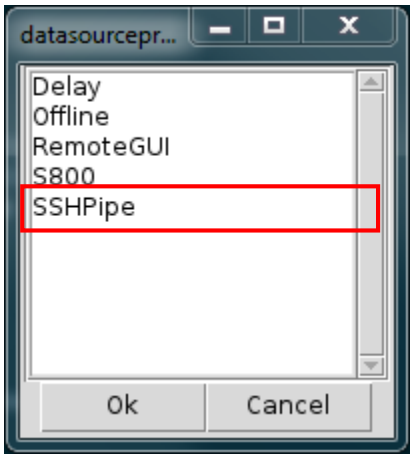
4.3.7 Register the Readout programs to the ReadoutGUI.

The ReadoutGUI is not associated with any Readout programs by default and needs to be provided the information concerning the ones that will form the system.

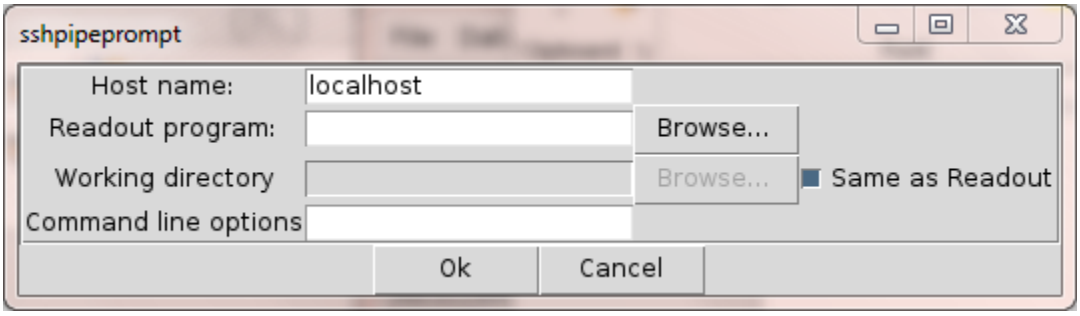
Select Data Source > Add… to add your first data provider.



The data provider dialogue will prompt you to choose the type of data provider you are going to use. For a VMUSBReadout, CCUSBReadout, or Readout program built with the SBS Readout Framework, select SSHPipe and then press the “Ok” button.



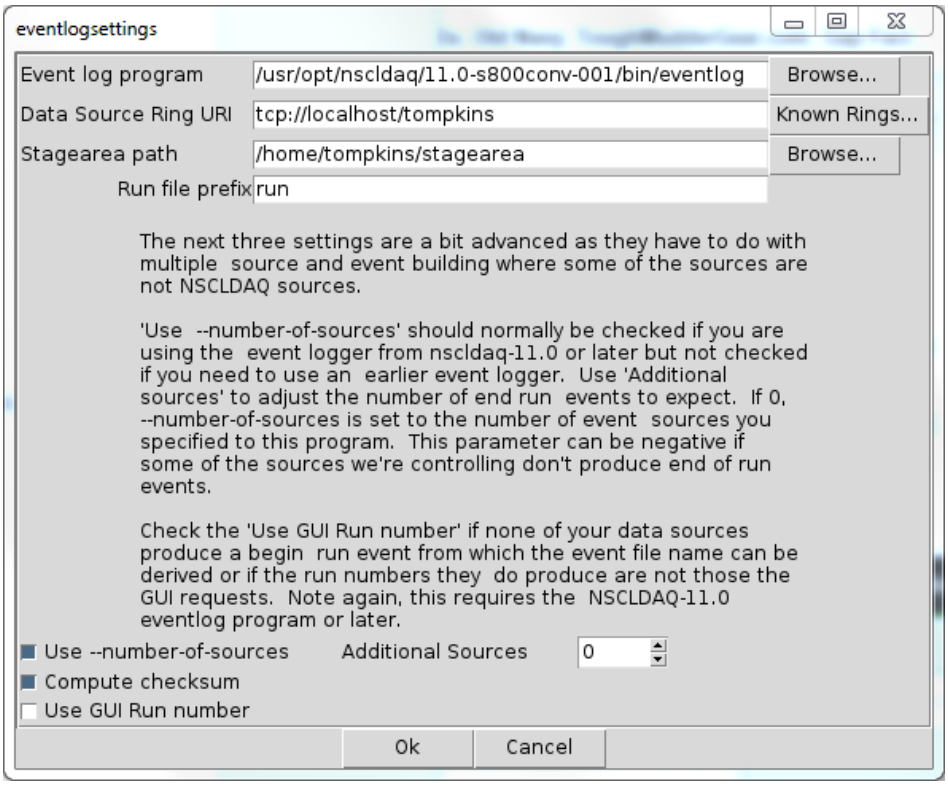
The data provider dialogue will prompt you to enter information for your first data source. Enter the path to your Readout program and any command line options that go with it. Press the “Ok” button.



Repeat this procedure for all of your basic sources. See Sections 5.1 and 5.15.3 for how to set up an S800 and RemoteGUI provider.

4.3.8 Configure Data Recording Parameters

Press the drop down menu button in Settings > Event Recording… You will be presented with the following dialog. For the most part, these should all be set correctly for a standard experiment. The most likely candidate that may need to be changed is the “Data Source Ring URI”. If you are running a single Readout program on a machine other than localhost, call it spdaq00, and it is filling a ringbuffer named “myring” on that computer, then you should enter tcp://spdaq00/myring. The second likely candidate to change is “additional sources”. See “Use –number-of-sources” and “Additional Sources” to know when to do so and what value to enter.



Once you have set up the various settings for the event recording, press the “Ok” button. Next, select the “Record data” checkbutton on the main window and then start and stop a run. You will know you have succeeded by verifying that the run files (.evt files) have been saved in the stagearea. For example, if the previous run was numbered 54, then in $HOME/stagearea/experiment there should now exist a directory named run54 with the following contents: run-0054-00.evt, run-0054.sha512, .started, and .exited.

4.3.8.1 Other options to customize event recording

The default values are pretty good for the event recording and are not likely to need changing. However, to address the many needs of experimenters, we provide the ability to customize. Think carefully about what you are trying to do if you intend to change these.

“Event log program”:

The program selected for this should match the data format being streamed. For example, if a Readout program based on NSCLDAQ 11.0 is being used, then the 11.0 eventlog program should be used. If instead a Readout program based on NSCLDAQ 10.2 is being used, then the 10.2 version of eventlog should be used.

“Stagearea path”:

Set this to be wherever your stagearea is. It defaults to $HOME/stagearea however, you can set it to be something else.

"Run file prefix":
It is best to leave this as "run". The eventlog program produces files named prefix-x-y.evt where prefix is selectable by this entry box, x is the run number, and y is the run segment.

"Use –number-of-sources" and "Additional Sources"
These are important when the eventbuilder and/or non-data-producing providers (NDPPs), such as "Delay", are being used. The number of additional sources, N_{add}, should be computed as:

$$N_{add} = N_{tot} - N_{reg} \ ,$$

where N_{tot} is the total number of Readout programs running in the entire DAQ and N_{reg} is the total number of data providers registered to the ReadoutGUI (including the NDPPs) in the previous step. Note that it is possible for this number to be negative.

"Compute checksum"
Deselecting this disables the calculation of an sha512 hash of the entire run file.

"Use GUI run number"
Typically the run number is computed from the data stream itself because the BEGIN_RUN contains that information. Selecting this checkbutton causes the run number present in the GUI to be assigned to the data irregardless of the run number specified within the data stream. In general, this is to be avoided as it can lead to confusion when analyzing data. However, it is made available for the very few instances when it is necessary.

If the event builder is not required, the experimental configuration is complete.

4.4 CONFIGURING THE EVENT BUILDER

It is assumed that at this point the ReadoutGUI has been configured to control one or more Readout programs. If this is not the case, then follow the steps in "Tutorial for common set up tasks:" before continuing. Attempting to use the event builder when the data sources are not configured properly will only complicate the setup process.

To best understand the following tutorial steps, some basic idea of how the event builder is constructed is useful to have. The event builder is a process that accepts data from an arbitrary number of client processes. Internally it stores data labeled with different source ids in separate queues. Each client process is responsible for consuming data items from a ring buffer, ensuring each has the appropriate information for processing, and then passing the products (i.e. fragments) to the event builder for processing. At startup, the event builder has no clients. For that reason, we have to register the clients to it. The following sections will instruct the reader on how to enable the event builder itself and then also how to set up and register clients.

4.4.1 Enabling the event builder add-on

The event builder add-on to the ReadoutGUI is enabled by adding some TCL commands to the ReadoutCallouts.tcl script. Until now, this script has not been mentioned. It is just a standard TCL script that is sourced once when ReadoutGUI is launched. To enable the use of the event builder, cut and paste the following code into your ReadoutCallouts.tcl script.

```
package require evbcallouts

::EVBC::useEventBuilder

proc OnStart {} {
  ::EVBC::initialize –restart false –glombuild true –glomdt 123 –destring evbout
}
```

At this point, the event builder is enabled and the next press of the "Start" button will append a bunch of TCL widgets to the bottom of the ReadoutGUI. The option values provided here specify that the event builder will not restart on each new run (-restart), event correlation will be enabled (-glombuild), events within 123 clock ticks of each other will be correlated (-glomdt), and the output will be put into the ring named evbout on localhost. If you prefer to change any of these parameters, feel free to do so here are via the controls made visible by the Start button.

4.4.2 Registering clients

The clients of the event builder are responsible for ensuring that the data items passed to it contain appropriate information for event building. These pieces of data are the source id, timestamp, and barrier type. Of these three elements, the reader need only concern himself/herself with the first two. The source id is an integer identifying which queue within the event builder will hold the data. This value is provided when registering the client and requires no setup. The timestamp, on the other hand, is a bit more tricky, because often the timestamp is a value read out from hardware each event. Because there is no way of knowing the logic for extracting this value, the reader must write some C/C++ code to define it. The following two steps instruct how to set up and register a single client. Simply repeat this for each client required. There will be a client for each stream of data the event builder is expected to merge.

There is a gotcha here that must be pointed out. ***If the user provided their Readout program with a timestamp extractor and source id, the data stream will already contain the information needed by the event builder to perform its tasks. For that reason, the source id and timestamp extractor provided when registering the client are ignored.*** *Despite this, you must still provide placeholder versions when registering the client. My recommendation is to copy the example extractor below into a file, compile it as is, and pass the resulting shared library when registering the client.*

4.4.2.1 Constructing a timestamp extractor

One of the things that needs to be provided when registering a client is a timestamp extract library. This is a dynamically linked shared library (ends in .so on linux) that contains a compiled function with the following signature:

```
uint64_t timestamp (pPhysicsEventItem item);
```

The function is passed a pointer to an entire physics event ring item and must return a valid 64-bit wide integer. It is here that the user must write C/C++ code to extract the timestamp value from the data. Because the logic for this is entirely experiment dependent, it is the user's responsibility. You can use the following boilerplate code to get started though.

```
#include <stdint.h>
#include <DataFormat.h>

extern "C" {

  uint64_t timestamp(pPhysicsEventItem item) {

    uint64_t tstamp = 0;

    if ( item->s_body.u_noBodyHeader.s_mbz == 0 ) {
      // data item does not have a timestamp.
      // a timestamp must be extracted and returned.

      uint32_t* body = reinterpret_cast<uint32_t*>(item->s_body.u_noBodyHeader.s_body);
      // add code here to compute and set tstamp
      // pointer "body" points to the beginning of the event data body.

    } // else timestamp is already present. No need to define an extractor.

    return tstamp;
  }

}
```

The structure of the pPhysicsEventItem, as well as all data formats, can be learned in $DAQROOT/include/DataFormat.h. Once you have implemented a satisfactory timestamp function, compile it into a shared library. If the file you created was named tstamp.cpp and the library you are going to produce is to be called libtstamp.so, then you can do so by typing the following at the command line:

```
% g++ -shared –o libtstamp.so –I$DAQROOT/include tstamp.cpp
```

If you are trying to do fancy things in your timestamp function that introduce dependencies on 3rd party code, this simple line may not suffice. However, extracting a timestamp is typically a simple process and can be accomplished without introducing the complexity that would make this simple compilation command fail.

4.4.2.2 Registering clients to the event builder

With a proper timestamp extractor library built, you are in a position to register your client. The way to do this is by calling the EVBC::registerRingSource proc for each client. These calls should follow after the EVBC::useEventBuilder call. For each of the data sources, add one of the following lines

```
::EVBC::registerRingSource sourceURI tstamplib id info
```

with the appropriate sourceURI, tstampLibPath, id, and info arguments. To illustrate how this works, consider an experiment in which the data produced by some silicon detectors are read out by a single Readout program in the system. The outputted data from this Readout are put into the “sidet0” ring buffer on localhost, and a timestamp can be properly extracted from each datum by the timestamp function recently compiled into the libtstamp.so shared library. Being the first client registered, we choose to label this stream of data with source id 0. We also want to associate a human readable description of “Si Dets” with it. In this scenario, the line that would be added to the ReadoutCallouts.tcl would be

```
::EVBC::registerRingSource tcp://localhost/sidet0 /path/to/libtstamp.so 0 “Si Dets”
```

Putting all of this together (and assuming you have another source to register), you might have the following ReadoutCallouts.tcl script.

```
package require evbcallouts

::EVBC::useEventBuilder

proc OnStart {} {
  ::EVBC::initialize –restart false –glombuild true –glomdt 123
}

EVBC::registerRingSource tcp://localhost/sidet0 /path/to/libtstamp.so 0 “Si Dets”
EVBC::registerRingSource tcp://localhost/gamma /path/to/libtstamp1.so 1 “Gamma Det”
```

5 APPENDIX

5.1 SAMPLE SOURCE CODE FOR XXUSBREADOUT TIMESTAMP EXTRACTORS

```
// rdotstamp.cpp
#include <stdint.h>
#include <assert.h>
#include <DataFormat.h>


extern "C" { // Do not mangle the names of the functions


  /**! Timestamp extractor for physics events
   *
   * A hook for the user to define the tstamp extraction algorithm for data
   * read by the event stack. The pointer provided points to the event header,
   * which is just the exclusive size of the event body in 16-bit units.
   *
   * This implementation expects that the 64-bits of data following the
   * event header are the tstamp.
   *
   * \param item  - pointer to the body of physics event (i.e. event header)
   *
   * \returns tstamp
   */
  uint64_t  getEventTimestamp(void* item)
  {
    // initialize the return value
    uint64_t time=0;

    // interpret the data as a bunch of 16-bit integers for sake of skipping
    // event header
    uint16_t* body = reinterpret_cast<uint16_t*>(item);

    // skip the event header (exclusive size of event in units of 16-bit words)
    body++;

    // The tstamp is the first 64-bits of the event, so recast the pointer
    // and dereference it to get the tstamp
    uint64_t* pTstamp = reinterpret_cast<uint64_t*>(body);
    time = *pTstamp;

    // done.
    return time;
  }



  /**! Timestamp extractor for scaler items
   *
   * Rather than explicitly produce a timestamp value from the data, this will
   * return 0. Timestamps with value 0 are treated specially by the event builder
   * because they are assigned the value of the last timestamped item. In that way,
   * they are just passed through the event builder in the order they were received.
```

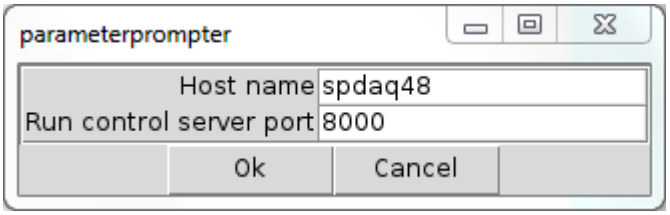
```
   * \param item  - pointer to the body of physics event (i.e. event header)
   *
   * \returns tstamp
   */
  uint64_t  getScalerTimestamp(void* item)
  {
    return 0;
  }

}
```

5.2 REGISTERING THE S800 AS A DATA PROVIDER

Registering the S800 to the ReadoutGUI is actually simpler than registering a standard Readout program as an SSHPipe. Here is how you do it.

1. Click on "Data Sources > Add…"
2. Select "S800" and press Ok.
3. You should then be prompted by a dialogue for a host and a port. Unless you know otherwise, enter "spdaq48" and 8000. When you are done, it should like this.



4. Press Ok.

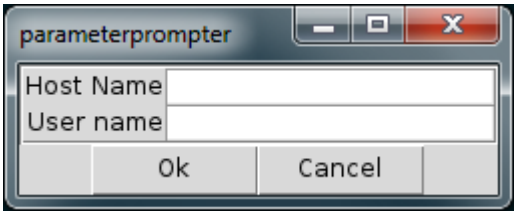
It is required that the S800DAQ is already running before you press the "Start" button. Otherwise, ReadoutGUI will try to establish communication with it and will never find it. It will fail miserably. So remember, start the S800DAQ first, then press "Start". Once you have pressed "Start", the S800DAQ will indicate on its GUI that it is in "slave" mode. What you have accomplished is only to have set up the run control operations. No data will flow out of the S800DAQ to an event builder.

5.3 REGISTERING A REMOTEGUI AS A DATA PROVIDER

First of all, what is a RemoteGUI provider? A RemoteGUI is a ReadoutGUI that is remotely controlled by a master ReadoutGUI. Every time the master ReadoutGUI transitions (e.g. beginning a run), the RemoteGUI is forced to transition as well. The capability this feature provides is modularization of the DAQ into subsystems, with each subsystem controlled by its own ReadoutGUI. The use case for this is when large detector systems must be able to operate independently for testing but then easily be integrated with other detector systems for an experiment. If each detector system has its own dedicated DAQ run by a ReadoutGUI, then integration requires just registering each of these ReadoutGUIs as a RemoteGUI provider. Here is how you do that:

1. Click on "Data Sources > Add…"
2. Select "RemoteGUI" and press Ok.

3. You should then be prompted by a dialogue for a hostname and a username. You should enter the name of the host that is connected to the hardware for that subsystem and the user name that was in use when starting the remote ReadoutGUI. The dialogue looks like this



4. Press Ok once you have entered your information.

Just like the S800DAQ, the ReadoutGUI to be controlled must be started already before pressing the "Start" button on the master ReadoutGUI. Also, the following lines need to be added to the ReadoutCallouts.tcl script of the remote ReadoutGUI.

```
package require ReadoutGuiRemoteControl
ReadoutGuiRemoteControl %AUTO%
OutputMonitor %AUTO%
```