# A Simple Experiment with VMUSBReadout

**Jeromy Tompkins**

**A Simple Experiment with VMUSBReadout**

by Jeromy Tompkins

Revision History

Revision 1.0 March 11, 2015 Revised by: J.R.T.
Original release

# Table of Contents

# List of Figures

# Chapter 1. Introduction

NSCLDAQ distributes with it a program called VMUSBReadout that supports the readout of the Wiener VM-USB VME controller. The following document will instruct the reader in how to set up a basic experiment consisting of a CAEN V785 peak sensing analog-to-digital converter, a CAEN V775 time-to-digital converter, and a Struck SIS3820 32-channel latching scaler. The level at which this tutorial is written assumes the following:

- No programming experience in Tcl
- Some familiarity with the C++ language
- Access to the required electronics
- Ability to wire up a signal processing circuit from a block diagram
- A version of NSCLDAQ 11 has been installed on the system in `/usr/opt/nscldaq/11.x-yyy`, where x is any minor version and yyy is any path version.
- The reader has already read the VMUSBReadout User's Guide found at the NSCLDAQ website (http://docs.nscl.msu.edu/daq) or in the `/usr/opt/nscldaq/11.0/share/pdfs` directory.

In addition to simply setting up the experiment, the user will be instructed at how to interpret the output of the devices using the dumper program for simple debugging and then on how to develop a basic SpecTcl application tailored to the specific experiment. The SpecTcl implementation will give an example of how to write an event processor using a framework independent data unpacker as well as supporting its potential reuse on event built data.

# Chapter 2. Setting up the Electronics

In this section, the reader will be given a schematic of the circuit the rest of this guide will assume. In order to set it up, the user must acquire *at least* the following:

- Wiener VM-USB controller
- CAEN V775 time-to-digital converter
- CAEN V785 peak-sensing analog-to-digital converter
- Struck SIS3820 32-channel scaler
- Gate and delay generator
- Delay module
- Constant-fraction discriminator
- Plenty of lemo cables
- Ribbon cable or a handful of twisted pair cables
- ECL-to-NIM converter
- Pulser
- VME crate
- NIM crate

Rather than detail how to plug each cable into each module, th electronics diagram for the digitizers is provided in *Block diagram of the V775 and V785 wiring*. Take note that this is a very basic circuit that makes no attempt to handle busy logic properly. It is designed specifically for generating some test data. Be aware also that there are some signal conversions not shown in the block diagram because their need depends on the exact hardware being used. It is left up to the reader to determine whether a logic signal needs to be converted from ECL to NIM or ECL to TTL or whatever between modules. A separate diagram is provided for the the scaler portion of the circuit at *Block diagram of SIS3820 scaler wiring*.

**Figure 2-1. Block diagram of the V775 and V785 wiring**

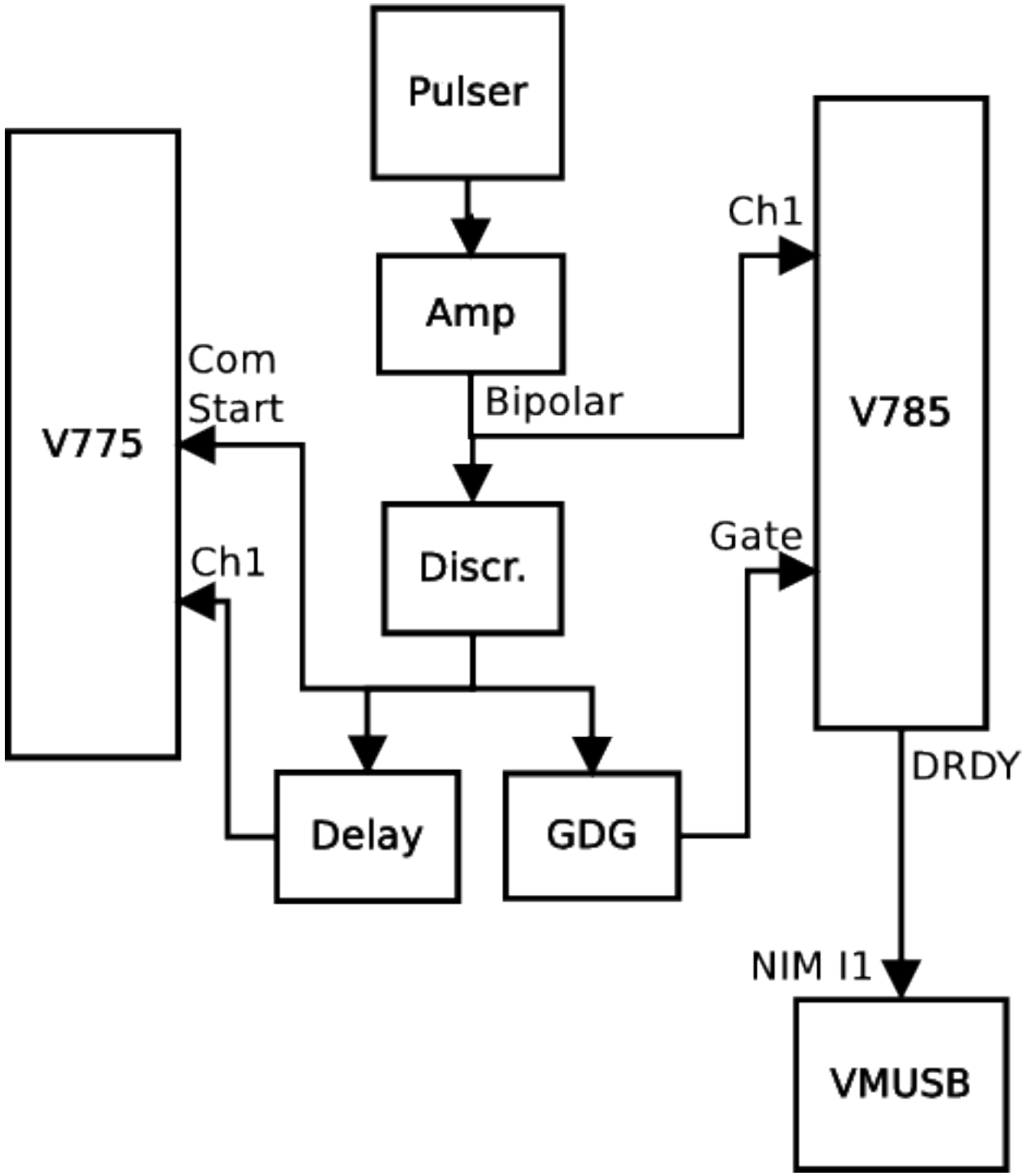
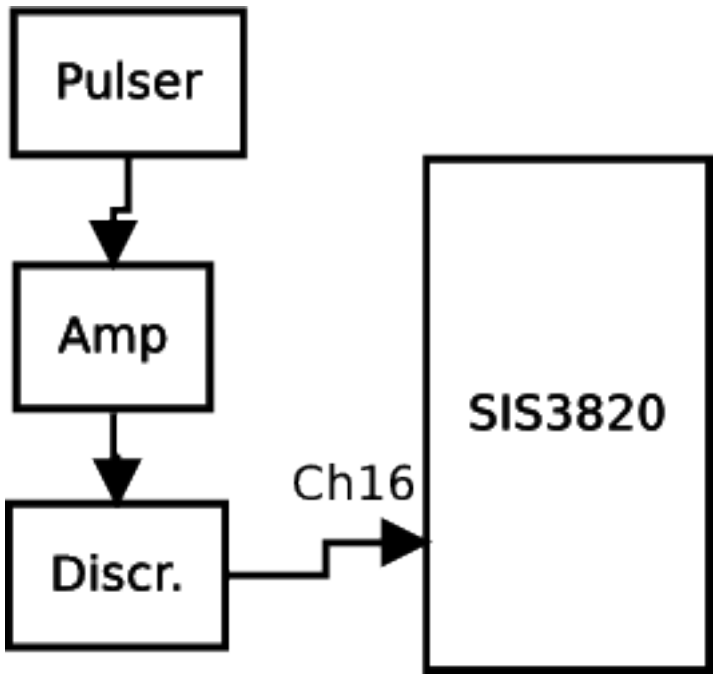


**Figure 2-2. Block diagram of SIS3820 scaler wiring**



# Chapter 3. The Configuration Files

Because we are using hardware that is already supported by VMUSBReadout, the only tasks left are to create the `daqconfig.tcl` and `ctlconfig.tcl` scripts.

The `ctlconfig.tcl` script is the configuration file for the slow-controls subsystem. In this situation, it will be an empty file because we are not attempting to use slow-controlled capabilities. We can quickly generate an empty file with the **touch** program at the command line. Here is how that works.

```
spdaqXX> touch ctlconfig.tcl
```

The next step is to build our `daqconfig.tcl` script. In that file, we will declare the V785, V775, and SIS3820 and add them to stacks. Doing so will cause them to be initialized at the beginning of each run, read out during triggers, and then transitioned back to an inactive mode at the end of each run. The V785 and V775 will be read out using a chained block transfer for each event trigger, and the SIS3820 will be periodically read using an internally generated trigger. We will therefore, need to define two separate stacks.

## 3.1. The event stack

We will begin with defining the stack that will read out our digitizer. Because this stack will read out the event data, we will refer to it as the "event stack". To begin setting up this stack, we have to first construct the device instances for the V785 and V775. Both of these conveniently share the same Tcl command ensemble, because they are essentially identical pieces of hardware. The command ensemble that create their device instances is called **adc**. It takes a set of options that are accepted for both V785 and V775 devices and then some more specialized options only valid for either the V785 and V775. The first module we will create is the V785. To do so we add the following lines to our `daqconfig.tcl` file:

```
set adcThresh [list 10 10 10 10  10 10 10 10 \
                    10 10 10 10  10 10 10 10 \
                    10 10 10 10  10 10 10 10 \
                    10 10 10 10  10 10 10 10 ]  ❶
adc create v785 0x11110000                      ❷
adc config v785 -geo 10 -threshold $adcThresh   ❸
```

❶ The V785 takes a list of 32 integers for its `-threshold` option. Each integer corresponds to the threshold for a channel. The first element of the list corresponds to channel 0 and the last to channel 31. For making our code more organized, we define a variable named `adcThresh` that will hold the list of 32 integers we created. Note that we created the list using a Tcl **list** command. This simply just returns a properly formatted Tcl list. The **set** command assigns the list to the variable.

❷ A new adc device instance is created with the name `v785`. The base address of the V785 in our crate is provided as the last argument to the line.

❸ The device instance named `v785` is configured. We specify via the `-geo` option which slot it lives in. This value will label the data outputted by the device during each readout cycle. It is important that this corresponds to the correct slot index the card is situated in. We also pass the list wof thresholds to the `-thresholds` option. The `$` syntax dereferences the variable `adcThresh` and returns the list that it refers to.

Using an an almost identical recipe, we will create a device instance for the V775. We want to define the time range of the TDC so we will configure the `-timescale` option.

```
set tdcThresh [list 10 10 10 10  10 10 10 10 \
                    10 10 10 10  10 10 10 10 \
                    10 10 10 10  10 10 10 10 \
                    10 10 10 10  10 10 10 10 ]
adc create v775 0x22220000                      ❶
adc config v775 -geo 11 -threshold $tdcThresh
adc config v775 -timescale 150                  ❷
```

❶ Just like the V785, we create the device instance using the **create** subcommand. The name we can refer to this device later in the script will be `v775`. The base address for the module is 0x22220000.

❷ The first line of configuration is almost identical to the V785 besides the different slot number and the different variable name holding the threshold list. The second line of configuration though establishes the time range of the TDC as a maximum 150 ns.

We now have two device instances defined in the configuration script. If we wanted to, we could immediately construct a stack using the **stack** command ensemble and pass our `v775` and `v785` device instance to it. However, we intend to read out our devices using a chained block transfer technique. A chained block transfer is an optimized means for reading out multiple devices. Instead of having to set up a block transfer from one device and then again from the second device, we can set up one block transfer and that reads out all of the devices in order. This functionality is implemented by the V785 and V775 rather than the VM-USB. To the VM-USB, this is just like a normal block transfer. To set this up in our `daqconfig.tcl` file we need to use the **caenchain** command. Here is how we do that:

```
caenchain create chain                  ❶
caenchain config chain -base 0x12000000 -modules [list v775 v785] ❷
```

❶ This line creates a device instance for the chained block transfer. We can refer to it by its name `chain` later in the file.

❷ The chained block transfer works by accessing the devices via a different address than their individual base addresses. In fact, by sending commands to this address, all of the modules who understand it can respond simultaneously. Some implementation require a multicast address as well as a chained block transfer address, the base address we provide for the Caen modules is dual purpose. It is important that only the top 8-bits of the base address provided here is used. Furthermore, we have also added the `v775` and `v785` device instances for chained readout. The order

of these modules should reflect the order in which they reside in the crate. The leftmost module in the crate should be the first module in the list and the rightmost module should be the last in the list.

The **caenchain** extends the idea of a device instance, because it does not really reflect a piece of hardware. Rather it is more of a virtual piece of hardware that handles the readout of multiple physical pieces of hardware. With it created and configured, we are ready to create our stack and we will only register the `chain` to the stack. Here is how that works

```
stack create evtStack
stack config evtStack -modules [list chain] -trigger nim1 ❶
```

❶ A single element list containing `chain` is added to the stack. Because our `v775` and `v785` are registered to the `chain` they will indirectly be read out. The `-trigger` option is passed a value of `nim1` to indicate that the stack will be triggered for execution whenever a logic pulse arrives at NIM input 1.

## 3.2. The scaler stack

With the event stack already defined, we now turn to the definition of the stack responsible for reading out the SIS3820. Because this stack will deal with scaler data, it will be referred to as the scaler stack. The procedure for setting up the scaler stack is the same as for the event stack. First we create a device instance for the SIS3820 and then we add it to a stack.

The Tcl command ensemble responsible for creating, configuring, and querying the state of an SIS3820 is named **sis3820**. There are two options that can be configured for the device instances it creates and we will only use the `-base` option. This will take the base address of the actual VME module. When this is added to a stack, it will cause the VM-USB to read all 32 channels of the device it is associated with. Let's add the following line to the bottom of our `daqconfig.tcl` file:

```
sis3820 create sclr 0x38000000
```

The scaler can then be added to a stack later by its name `sclr`. We will do that right now. Creating the stack for the scaler readout is done as normal using the **stack**. However, to define this a scaler stack that gets periodically read out we need to specify different values for the options.

```
stack create sclrStack
stack config sclrStack -modules [list sclr] -trigger scaler -period 2 ❶
```

❶ Purposing a stack for scaler readout is accomplished by passing `scaler` for the `-trigger` argument. Not only does this enable periodic execution of the stack, but it also labels the stack as index 1. VMUSBReadout will treat the data read by this stack specifically as scaler data and turn it

into ring items of type PERIODIC_SCALERS. The `-period` option was passed a value of 2 to specify that we want the readout period to be 2 seconds.

The scaler stack is now complete.

# Chapter 4. Running VMUSBReadout

There is a detailed overview of how to run VMUSBReadout in the comprehensive documentation of NSCLDAQ. In this section we will assume that that has been read already and we will proceed by defining a simple launcher script.

Unless you intend to keep your configuration scripts in the directory `~/config`, you are required to specify the location of the configuration scripts at launch. To avoid this, we will generate a short bash script that will launch VMUSBReadout with the local configuration scripts. To do so, we create a script called `govmusb` that has the following contents.

```
#!/bin/sh                                                                    ❶

$DAQBIN/VMUSBReadout --daqconfig=$PWD/daqconfig.tcl --ctlconfig=$PWD/ctlconfig.tcl ❷
```

❷ We assume that the DAQBIN environment variable has been specified by sourcing the `daqsetup.bash` script. If you have not done this, you can source the script by doing (at the NSCL):

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

The `PWD` environment is defined by default and stands for "present working directory." Note that this implies that the configuration scripts must live in the directory that the `govmusb` script is executed from.

To make the script executable, we have to change the permissions. Unless you want to prevent other people from executing the launcher script you can let all users execute it. To do that we use the **chmod** command.

```
spdaqXX> chmod a+x govmusb
```

At this point you should be able to start up VMUSBReadout by executing the script. That is done by doing:

```
spdaqXX> ./govmusb
```

At the prompt of VMUSBReadout, you can use the **begin**, **end**, **pause**, and **resume** commands to start, stop, pause, and resume runs.

# Chapter 5. Understanding the Output

The output of the program can be inspected by attaching the dumper program to the ring buffer receiving the data outputted from VMUSBReadout. Because we did not provide a different ring buffer at the command line when launching our program, the output will go to a ring buffer whose name is the same as your username. This is actually the default value of the `--source` command line option of the dumper program. As a result we can attach the dumper with the following command from a different terminal than the one running VMUSBReadout.

```
spdaqXX> dumper
```

If the run is active, you should see data printing to your screen. You can kill the dumper program by pressing `CTL-C`. A cleaner way to do this in the future is to provide dumper with a finite number of ring items to process. For example, to process 10 ring items and then exit, one would enter:

```
spdaqXX> dumper --count=10
```

## 5.1. Event data

However you do this, you should see event output that might look like this:

```
------------------------------------------------------------
Event 34 bytes long
Body Header:
Timestamp:    5764974207242862948
SourceID:     0
Barrier Type: 0
0010 0100 5200 4d7f 5001 0c08 5400 0100
5a00 4de4 5801 0c05 5c00 ffff ffff ffff
ffff
------------------------------------------------------------
```

From the VMUSBReadout user's guide, we know that the first 16-bit word is the event header produced by the VM-USB itself. The most-significant four bits specify that the data corresponds to stack 0 and the least significant 12 bits specify that there are sixteen 16-bit words that follow. This is very sensible because we only defined a single event stack and we can count the remaining data words.

The remaining 16 words must be understood according to the data format of the V785 and V775. Each of these devices outputs a series of 32-bit words bookended by header and end-of-event words. Bits 24-26 uniquely identifies betweeen those two words and the data words. The format also uses the most

significant five bits (bits 27-31) for the geographic address. The geographic address should be the same value we provided for the `-geo` option. For that reason, we expect that bits 27-31 of each 32-bit word can be used to identify whether the data originated from the V775 or the V785. Before we start parsing the data further, let's group the 16-bit words in the output into 32-bit words.

```
0x0010          ❶
0x52000100      ❷
0x50014d7f      ❸
0x54000c08      ❹
0x5a000100      ❺
0x58014def      ❻
0x5c000c05      ❼
0xffffffff
0xffffffff      ❽
```

❶ VM-USB event header. This is just 16-bits.

❷ Header word for the V775 identified by bits 24-26 being the value 2. The most-significant 5 bit of this number is 10, which matches what was defined for the `-geo` option of the v775. Other information encoded in this header are the crate index (bits 16-23) and the number of data words before the end-of-event word (bits 12-16). The header tells us that the module resides in crate 0 and that there is a single data word that follows.

❸ This is the lone data word for the V775 identified by bits 24-26 being 0. Bits 27-31 still contain a geographic address of 10 as it should. Bits 16-20 identify the channel number for the data, bits 0-11 identify the value and bits 12 and 13 are the underflow and overflow bits. We can use this information to understand that digitized value did not underflow or overflow and resulted in the value 3455.

❹ As we expect by header word of the V775, this is the end of event word. We know this for sure because bits 24-26 store the value 4. Once again the bits 27-31 store the value 10 so the word originated from the V775. The lower 24 bits store the event count, which converts to 3080 in decimal.

❺ This word is the first word of the V785. It is a header word that corresponds to slot 11 and crate 0. There is one data word that follows prior to the end-of-event word.

❻ Here is the data for the V785. It corresponds to channel 1 and has a value f 3567.

❼ Here is the end-of-event word for the V785. It corresponds to event number 3077.

❽ The 0xffffffff is what is returned by the devices when they complete their block transfer. There are two of these because each module is emitting BERR to signify they are done being read out.

In the above output, it is apparent that the event count of the V775 and V785 differ by 2 events, the V775 having seen two more events than the V785. Such a condition is not terribly worrisome in this case, because the V775 is initialized prior to the V785. Because the initialization process is done before the VM-USB transitions to autonomous mode, the timing is controlled by the speed at which a VM-USB can execute interactive commands and the speed at which the software operates. Such timing is indeterminate because the operation system scheduler plays a role. It should be expected that the time

between clearing the V775 and V785 is on the order of milliseconds. If the rate is around 1 kHz, then it is possible that multiple triggers occur after the V775 has been cleared and the V785 clears.

## 5.2. Scaler data

If your most recent dump of the data using dumper did not include any scaler items in the output, you should run the dumper again in such a way that excludes processing ring items of type PHYSICS_EVENT.

```
spdaqXX> $DAQBIN/dumper --count=5 --exclude=PHYSICS_EVENT
```

The output should include at least one ring item that has the following look:

```
Tue Mar 10 15:12:56 2015 : Scalers:
Interval start time: 10 end: 12 seconds in to the run

Body Header:
Timestamp:    0
SourceID:     0
Barrier Type: 0
Scalers are incremental
Index         Counts                Rate
    0              0                0.00
    1              0                0.00
    2              0                0.00
    3              0                0.00
    4              0                0.00
    5              0                0.00
    6              0                0.00
    7              0                0.00
    8              0                0.00
    9              0                0.00
   10              0                0.00
   11              0                0.00
   12              0                0.00
   13              0                0.00
   14              0                0.00
   15              0                0.00
   16            513                256.50
   17              0                0.00
   18              0                0.00
   19              0                0.00
   20              0                0.00
   21              0                0.00
   22              0                0.00
   23              0                0.00
   24              0                0.00
```

```
   25              0               0.00
   26              0               0.00
   27              0               0.00
   28              0               0.00
   29              0               0.00
   30              0               0.00
   31              0               0.00
-------------------------------------------------------
```

There is much less to be explained here than for the event data. The scaler stack read out all 32 channels of data from the SIS3820 during each stack execution, so the dump shows 32 channels of data. In my test setup, I had only plugged in a single input to the SIS3820 at channel 16. The SIS3820 was operated as an incremental scaler as well, which means that after every read the device was cleared. In the information section above the actual scaler values, you also see that the scaler values account for the time between 10 and 12 seconds after the run began.

# Chapter 6. Developing a Tailored SpecTcl

The dumper is not the most useful tool for understanding the data read out by the electronics. Instead we should be using SpecTcl for that purpose, because it provides a much more straightforward and intuitive way to inspect data. To use SpecTcl, we have to teach it how to retrieve the salient features of our data and store them as tree parameters. Tree parameters are objects that behave as though the are plain old double values. What makes them special is that they are histogrammable entities. By unpacking raw data into tree parameters, we can use SpecTcl to quickly define histograms from them.

An example of a simple SpecTcl implementation for unpacking a single V775 exists in conjunction with the SBS Readout framework. To not repeat what has already been demonstrated, this will demonstrate how to develop a SpecTcl whose parsing utilities are separated from the SpecTcl framework. In principle, the parsing class used here could be reused in other analysis framework like ROOT. It will also demonstrate a modern style of implementing C++ that leverages some newer C++11 features, like the range-based for loop.

## 6.1. Acquiring the Skeleton

There is a good deal of boilerplate code that goes into developing a tailored SpecTcl. To avoid rewriting a lot of that code, you can start from the "skeleton" implementation. This provides a fully functional SpecTcl application that can be easily modified to support our specific needs. Getting the skeleton can be achieved by doing:

```
spdaqXX> mkdir MySpecTcl
spdaqXX> cd MySpecTcl
spdaqXX> cp /usr/opt/spectcl/3.4/Skel/* .
```

It does not matter all that much which specific version is used so long as it is at least version 3.4. Prior to version 3.4 there was no support for the NSCLDAQ 11.0 data format. Because our data has been acquired using an 11.0 version VMUSBReadout, it is therefore not possible to analyze it using an older version of SpecTcl.

## 6.2. Writing our Event Processor

SpecTcl has an analysis engine in it that handles all of the input/output type operations for the user. It can do so because it understands how to read raw data from an input stream and parse it into entities of a given data format. Once it has determined the type of entity, it passes it to an analysis pipeline for processing. The details of the processing is very experiment specific and must be provided by the experimenter. This is done by deriving a new class from the EventProcessor class and then registering it to the pipeline. Our derived EventProcessor will get passed the beginning of each physics event body for processing.

We intend to write an event processor that clearly separates the SpecTcl-like dependencies from the unpacking code. It will be named CRawUnpacker because it will operate on the raw data format. The CRawUnpacker will be responsible for doing SpecTcl related things, like storing data into tree parameters, using a SpecTcl independent unpacking routine. The latter will be a class named CRawADCUnpacker that will parse the data format produced by the V785 and V775 and store the resulting information in ParsedADCEvent objects.

It is good practice in C++ to declare the capabilities of our class in a header file and then implement those capabilities in a source file. Doing so makes the code more flexible, reusable, and less bloated. We will follow this practice while implementing our SpecTcl, which means we have four files to create.

## 6.2.1. CRawUnpacker

Without further ado, let's start by defining our CRawUnpacker header file, `CRawUnpacker.h`

```
#ifndef CRAWUNPACKER_H
#define CRAWUNPACKER_H    ❶

#include <config.h> ❷
#include "CRawADCUnpacker.h"
#include <EventProcessor.h>
#include <TreeParameter.h>
#include <cstdint>  ❸
#include <cstddef>  ❹

class CEvent;
class CAnalyzer;
class CBufferDecoder;     ❺

class CRawUnpacker : public CEventProcessor ❻
{
  private:
    CRawADCUnpacker      m_unpacker;
    CTreeParameterArray  m_values;   ❼

  public:
    CRawUnpacker();
    virtual ~CRawUnpacker();

    virtual Bool_t operator()(const Address_t pEvent,
                              CEvent& rEvent,
                              CAnalyzer& rAnalyzer,
                              CBufferDecoder& rDecoder);  ❽
  private:
    Bool_t unpack(TranslatorPointer<std::uint32_t> begin,
                  std::size_t nLongWords);   ❾
};

#endif
```

❶ To protect against including the header twice, we add these preprocessor directives to only include it once. This is called an include guard.

❷ The `config.h` file contains lots of preprocessor definitions. You must include it prior to other files.

❸ To use the `std::uint32_t` type, we need to include this header. We choose to use this over `stdint.h` because it is more portable.

❹ To use the `std::size_t` type, we need to include this header.

❺ Because some of the arguments in the methods of our class will take references to objects, we need to forward declare their types. This does not define the classes, it just introduces the name of these classes as a valid type. We will have to actually include definitions later if we are going to use the functionality of these types.

❻ Our CRawUnpacker class derives from the CEventProcessor base class. This makes it possible to use the event processor in the SpecTcl event processing pipeline.

❼ The event processor will maintain an object that handles the parsing of the data itself in a platform independent way. The result of the parse will be used to set tree parameters. The `CTreeParameterArray` is a convenient collection of tree parameters.

❽ The operator() method is called for every ring item of type PHYSICS_EVENT that is read from the input stream of data.

❾ The method that actually handles initiation of parsing and setting tree parameters with the results. The first argument of this is a smart pointer that transparently handles byte swapping. It behaves like a pointer, dereferencing it returns a uint32_t.

The implementation of our CRawUnpacker class will be found in the source file `CRawUnpacker.cpp`. This is naturally a bit longer of a file, because it includes the implementation code. The constructors and destructors are nearly trivial and will be dealt with in one fell swoop.

```
#include "CRawUnpacker.h"
#include <BufferDecoder.h>
#include <TCLAnalyzer.h>
#include <iostream>
#include <stdexcept>

using namespace std; ❶

CRawUnpacker::CRawUnpacker()
  : m_unpacker(),
    m_values("t",4096,0.0,4095.0,"channels",64,0) ❷
{
}

CRawUnpacker::~CRawUnpacker()
{
}
```

❶ This brings uint32_t, size_t, and exception into scope. We avoid having to prefix these types with `std::` every time.

❷ The list of comma separated calls following the colon are called an initialize list. We use it to construct the data members. The second element of the list create a `CTreeParameterArray` consisting of 64 tree parameters, each having 4096 bins in a range of 0 to 4095. The names given to these parameter are of the form "t.XX" where XX is the index, left-padded with zeroes.

The more interesting code occurs in the operator() and unpack() methods. The operator() method is responsible for the actual processing of the data as well as telling SpecTcl what the size of the event is. It will just handle the calculation and reporting of the event size and then delegate the parsing and handling of the results to the unpack() method.

> **Tip:** At least one event processor is required to set the event size and to avoid introducing confusing bugs into your SpecTcl, it is best practice to only do this in the first event processor of the pipeline.

The resulting operator() method will look like this:

```
Bool_t
CRawUnpacker::operator()(const Address_t pEvent,
                         CEvent& rEvent,
                         CAnalyzer& rAnalyzer,
                         CBufferDecoder& rDecoder)
{
  TranslatorPointer<uint16_t> p(*rDecoder.getBufferTranslator(), pEvent);  ❶

  CTclAnalyzer& a(dynamic_cast<CTclAnalyzer&>(rAnalyzer)); ❷

  size_t  size = (*p++ & 0x0fff); ❸

  // the event header is exclusive so the actual size of the full event
  // is actually the
  a.SetEventSize((size+1)*sizeof(uint16_t)); ❹

  size_t nLongWords = size*sizeof(uint16_t)/sizeof(uint32_t); ❺

  return unpack(p, nLongWords);
}
```

❶ The data is not gauranteed to have native byte ordering because it may have originated on a different host. SpecTcl provides a smart pointer called TranslatorPointer that handles the byte-swapping automatically. Here we create one that refers to the buffer managed by the buffer decoder at the address stored by pEvent. This is also a way to convert the pEvent to a more useful type than Address_t.

❷ The standard implementation of SpecTcl used at the NSCL makes use of a CTclAnalyzer. The CTclAnalyzer needs to be told how big the buffer is in order to successfully traverse the data stream. A method to set the event side is available in CTclAnalyzer but not in the generic CAnalyzer class. For that reason, we upcast the reference passed into our analyzer. If this is invalid, dynamic_cast will

throw an exception of type std::bad_cast. We know it will succeed though because our analyzer is a CTclAnalyzer. Note that we cast to a reference to avoid copying the analyzer unnecessarily.

❸ The first 16-bit word of the VM-USB buffer contains the event header. The entire size of the body less 1 is contained in the lower 12-bits of this word. We use a bitwise-AND operator to extract that number. Note that the `*p++` effectively retrieved the value pointed and then incremented the pointer forward by 16-bits.

❹ Here we pass the size of the event to the analyzer. The argument requires a size in units of bytes rather than 16-bit words. We convert by multiplying our number by the number of bytes in a 16-bit word (`sizeof(uint16_t)`). Note that we could have just multiplied by 2, but chose to do otherwise for readability sake.

**Tip:** Write for people to understand your code easier. Code is written once and read everytime thereafter.

❺ The remainder of the buffer referred to by our pointer `p` is composed of 32-bit integers. We need to know how many of these there are and will compute it. We are converting our size from units of 16-bit integers to units of 32-bit integers.

The unpack() method implementation will look like this:

```
Bool_t
CRawUnpacker::unpack(TranslatorPointer<uint32_t> begin,
                     size_t nLongWords)
{
  auto end = begin+nLongWords;  ❶

  try {  ❷
    vector<ParsedADCEvent> events = m_unpacker.parseAll(begin, end);  ❸

    int offset = 0;
    for (auto& event : events) {  ❹
      offset = (event.s_geo-10)*32;  ❺
      for (auto& chanData : event.s_data) {  ❻
        m_values[chanData.first+offset] = chanData.second;
      }
    }
  } catch (exception& exc) {  ❼
    cout << "Parsing Failed! Reason=" << exc.what() << endl;
    return kfFALSE;  ❽
  }

  return kfTRUE;  ❾
}
```

❶ The `auto` keyword is a shortcut introduced in C++11 that causes the compiler to deduce the type of `end`. Because `begin` is of type `TranslatorPointer<uint32_t>`, `end` will be of the same type but will point to a location in memory `nLongWords` away.

❷❼❽ The CRawADCUnpacker::parseAll() method may through an exception and this along with the `catch` ensure that any object thrown of type derived from `std::exception` will be caught. We choose to print a message to the user and then return false. Returning this causes the event to be excluded from the histogramming process.

❸ Here is where we call our parser. It will return a list of parsed events if it succeeds.

❹ This is another C++11 construct called a range-based for loop. Basically what this does is loop through each element of the `events` vector and assign a reference called `event` to it. We are using it to access the data of the V775 and then the V785.

❺ The channel data of the V775 and V785 are stored in the same array of tree parameters. We compute an offset based on the slot number of the module. The data labeled with slot number (`s_geo`) 10 will be at offset 0, whereas those from slot 11 will be at offset 32.

❻ Here we use another range-based for loop to iterate through the channel data of a specific digitizer's data.

❾ If we reached this point, our entire event was parsed successfully and it is sensible to histogram the results. We indicate this by returning true.

That is it for the implementation ofthe CRawUnpacker class. Before moving on, I would like to explain the motivation for the unpack() method. It is not obvious why its contents could not have just been included as the body of the operator() method. They very well could have and it would have worked perfectly fine for our specific setup. However, splitting it off makes our event processor much more reusable.

One of the reasons NSCLDAQ 11.0 was created was for better support of event building. The very presence of the capability makes it likely that we might want to use it. Separating the the unpack() from the operator() is aimed at supporting this scenario. The event builder outputs data that looks a bit different than the input data it receives, because it appends extra information to each item and then groups those together. In the lingo of the event builder, PHYSICS_EVENT ring items that enter the event builder are transformed into fragments that are glommed into built ring items of type PHYSICS_EVENT. The built ring item has a body stuffed with fragments. If we tried to parse the body of one of these using our operator() as it is currently implemented, SpecTcl would fail miserably. However, the same data that this parser understands is present in that built ring item. It is just buried deeply in a fragment. So the logic in this unpacker is still useful if we can traverse through the structure of the built body. By separating the unpack() method from the operator() method, we can call it once the right part of the data has been found. You might be wondering why we don't call the operator() method. The reason is that the operator method is responsible for reporting the event size. In a built ring item, the value this would extract would be incorrect. The unpack() method allows us to bypass it.

## 6.2.2. The CRawADCUnpacker Class

The CRawUnpacker depends on the presence of a parser class for the V785 and V775 class. That will be

implemented in the CRawADCUnpacker class. I had stated that I would do this in a framework independent way. I will do so by showing a simpler example that depends on the TranslatorPointer<uint32_t> type for simplicity and then will explain how to generalize this afterwords. This will keep the example accessible to more people that don't need their code 100% independent but will also provide a clear path forward for those who do need it.

We will begin by describing the problem this class will solve. In the most fundamental sense, we have to work our way sequentially through the data, identify each piece, store the information in a framework independent entity, and identify when we are done. By doing this we will in essence validate the structure of the data. If it is different than we expect, then we cannot gaurantee that we understand it and will fail. If we do not encounter an error, we will return our platform independent type filled with the parsed information. Let's start by defining that type, which we will call ParsedADCEvent.

```
#ifndef CRAWADCUNPACKER_H
#define CRAWADCUNPACKER_H

#include <vector>
#include <utility>
#include <cstdint>
#include <TranslatorPointer.h>

struct ParsedADCEvent
{
  int s_geo;
  int s_crate;
  int s_count;
  int s_eventNumber;
  std::vector<std::pair<int, std::uint16_t> > s_data; ❶
};

#endif
```

❶ The digitized value and channel it is associated with group naturally together. We accomplish this by storing the two values as a `std::pair`, which is just a container. The first element of the pair is the channel index and the second element is the digitize value. Because a single event may consist of up to 32 digitized values, the ParsedADCEvent maintains a vector of these.

The ParsedADCEvent contains all of the information that we care to keep from a complete event. The data for each channel will be stored as a pair in a vector. The first element of each pair will be the channel number while the second element will be the digitized value.

The next thing we need to do is define the CRawADCUnpacker class itself. Two entry points will be provided for public use. One of them is used by the CRawUnpacker class, parseAll(), and it iteratively calls the second, parseSingle(). We will focus primarily on the implementation of parseSingle() because it is the true parsing routine. The parseSingle() method depends on a handful of helper methods that will be explained as they come up. Here is the class declaration of the CRawADCUnpacker. In

`CRawADCUnpacker.h`, it would follow the end of the ParsedADCEvent definition but precede the `#endif` preprocessor directive.

```
class CRawADCUnpacker
{
  public:
    using Iter=TranslatorPointer<std::uint32_t>; ❶

  public:
    std::vector<ParsedADCEvent>
      parseAll(const Iter& begin, const Iter& end); ❷

    std::pair<Iter, ParsedADCEvent>
      parseSingle(const Iter& begin, const Iter& end);

  private:
    // Utility methods
    bool isHeader(std::uint32_t word);
    bool isData(std::uint32_t word);
    bool isEOE(std::uint32_t word);

    void unpackHeader(std::uint32_t word, ParsedADCEvent& event);
    void unpackDatum(std::uint32_t word, ParsedADCEvent& event);
    Iter unpackData(const Iter& begin, const Iter& end, ParsedADCEvent& event);
    void unpackEOE(std::uint32_t word, ParsedADCEvent& event);
};
```

❶ To save from repeatedly writing the type name `TranslatorPointer<std::uint32_t>` over and over again, we alias it as `Iter`. This is just another way to define a typedef. I will explain later how this also helps loosen the dependence of the class on the SpecTcl framework.

❷ This and the next method form the public interface for our parse. Other code can either parse a chunk of ADC data as a series of subsections from multiple devices or a single section from a single device.

We know that the data from the ADC must come in a specific order. The event header should come first followed by a sequence of data words, and then ultimately an end of event or trailer word. Furthermore, the header word is descriptive about how many data words exist so by reading it we will know when to expect the end of event word. If the data we are parsing violates this structure, then something bad has happened. Each stage will check whether the data is valid and throw if it is not. Here is the code for the parseSingle() method and includes that we need to add to our `CRawADCUnpacker.cpp` file:

```
#include "CRawADCUnpacker.h"
#include <string>
#include <stdexcept>
#include <iostream>

using namespace std;
```

```
static const uint32_t TYPE_MASK (0x07000000);
static const uint32_t TYPE_HDR  (0x02000000);
static const uint32_t TYPE_DATA (0x00000000);
static const uint32_t TYPE_TRAIL(0x04000000);

static const unsigned GEO_SHIFT(27);
static const uint32_t GEO_MASK (0xf8000000);

static const unsigned HDR_COUNT_SHIFT(8);
static const uint32_t HDR_COUNT_MASK (0x00003700);
static const unsigned HDR_CRATE_SHIFT(16);
static const uint32_t HDR_CRATE_MASK (0x00ff0000);

static const unsigned DATA_CHANSHIFT(16);
static const uint32_t DATA_CHANMASK (0x001f0000);
static const uint32_t DATA_CONVMASK (0x00003fff);

static const uint32_t TRAIL_COUNT_MASK(0x00ffffff);
static const uint32_t BERR(0xffffffff); ❶


pair<CRawADCUnpacker::Iter,ParsedADCEvent>
  CRawADCUnpacker::parseSingle(const Iter& begin, const Iter& end)
{

  ParsedADCEvent event;             ❷

  auto iter = begin;                ❸
  if (iter<end) {                ❹
    unpackHeader(*iter++, event); ❺
  } else {
    string errmsg("CRawADCUnpacker::parseSingle() ");
    errmsg += "Incomplete event found in buffer.";
    throw runtime_error(errmsg);
  }

  int nWords = event.s_count;
  auto dataEnd = iter+nWords;

  if ((dataEnd > end) || (dataEnd == end)) { ❻
    string errmsg("CRawADCUnpacker::parseSingle() ");
    errmsg += "Incomplete event found in buffer.";
    throw runtime_error(errmsg);
  } else {
    iter = unpackData(iter, dataEnd, event); ❼
  }

  if (iter<end) {              ❽
    unpackEOE(*iter++,event);     ❾
  } else {
    string errmsg("CRawADCUnpacker::parseSingle() ");
    errmsg += "Incomplete event found in buffer.";
    throw runtime_error(errmsg);
```

```
  }

  return make_pair(iter,event);      (10)
}
```

❶ This and the lines before it define useful bitmasks and shifts for performing bitwise arithmetic on the data. We will use them to extract the pieces of information from each word.

❷ Here we create a brand new, empty ParsedADCEvent that will be filled with the parsed data.

❸ We create a new translator pointer to iterate over the body of data.

❹❽ Every time that we move the iterator forward, we need to make sure that we do not step out of bounds. Dereferencing an iterator beyond its bounds of validity is asking for all kinds of pain.

❺ The logic for parsing the header lives in a utility method.

❻ Just because the header word tells us that a certain number of data words should follow does not mean they actually do. This ensures that the data we expect is there before accessing it.

❼ A utility method is called to handle the unpacking of all the data words. The resulting location of the buffer following the last data word is returned by the method. It is used to move the iterator to a new position.

❾ A utility method handles the parsing of the end-of-event word.

**(10)** This method must both pass the next unprocessed location of the buffer back to the caller in case it desires to continue parsing as well as the newly parsed event data. We do so by using a function to create an object of type `std::pair<CRawADCUnpacker::Iter,ParsedADCEvent>`.

As you can see, there are many supporting pieces. We will look at them one by one, beginning with the unpackHeader() method.

```
bool CRawADCUnpacker::isHeader(uint32_t word)
{
  return ((word&TYPE_MASK)==TYPE_HDR);    ❶
}


void CRawADCUnpacker::unpackHeader(uint32_t word, ParsedADCEvent& event)
{
  if (! isHeader(word) ) {
    string errmsg = "CRawADCUnpacker::parseHeader() ";
    errmsg += "Found non-header word when expecting header. ";
    errmsg += "Word=";
    errmsg += to_string(word);
    throw runtime_error(errmsg);
  }   ❷

  event.s_geo   = ((word & GEO_MASK)>>GEO_SHIFT);
  event.s_crate = ((word & HDR_CRATE_MASK)>>HDR_CRATE_SHIFT);
  event.s_count = ((word & HDR_COUNT_MASK)>>HDR_COUNT_SHIFT);  ❸
}
```

❶ Here we use bitwise arithmetic to check that bits 24-26 are set to the value 2.

❷ If the word is not a header word, we failed to properly understand the data and parsing should stop immediately. An exception is thrown.

❸ Using bitwise arithmetic again, we extract the geo value (slot number), crate number, and number of data words to follow. These are all stored to the event we are filling.

The data words are unpacked in a bit more fancy method called unpackData(). It loops through a range of words designated by its arguments, calling unpackDatum() on each one. The unpackDatum() is very similar to unpackHeader() method in that it extracts specific pieces of data from the data word it is passed. The difference is that the values it extracts for the channel and value are paired together and stored in the vector of the event. This is what those look like:

```
bool CRawADCUnpacker::isData(uint32_t word)
{
  return ((word&TYPE_MASK)==TYPE_DATA);  ❶
}


void CRawADCUnpacker::unpackDatum(uint32_t word, ParsedADCEvent& event)
{
  if (! isData(word) ) {
    string errmsg = "CRawADCUnpacker::unpackDatum() ";
    errmsg += "Found non-data word when expecting data.";
    throw runtime_error(errmsg);
  }

  int channel   = ((word & DATA_CHANMASK)>>DATA_CHANSHIFT);
  uint16_t data = (word & DATA_CONVMASK);   ❷

  auto chanData = make_pair(channel,data);
  event.s_data.push_back(chanData);             ❸
}


CRawADCUnpacker::Iter
CRawADCUnpacker::unpackData(const Iter& begin,
                            const Iter& end,
                            ParsedADCEvent& event)
{
  // only allocate memory once because we know how much we need already
  event.s_data.reserve(event.s_count);       ❹

  auto iter = begin;
  while(iter != end) {

    unpackDatum(*iter, event);               ❺

    ++iter;
```

```
  }

  return iter;    ❻
}
```

❶ We check that bits 24-26 store the value 0.

❷ Using bit-wise arithmetic we extract the channel and value.

❸ The channel and value are paired together and then added to the end of the vector.

❹ Initially the vector that will store the channel data has zero size. Every time that a new element is added we will push it onto the back of the vector. By default, this will likely cause the vector's memory buffer to be reallocated everytime we add an element, which is costly. Because we already know the final size, we can allocate the total memory once using the `vector<>::reserve` method.

❺ To parse all of the data words, we just iterate through the range that has been passed in as parameters. Each word is unpacked by a call to the unpackDatum() method. If the word is not a data word, an exception is thrown.

❻ The location of the next unparsed word in the buffer is returned to the caller.

Finally, we reach the unpackEOE() method. Once again, this is extremely similar to the unpackHeader() method, moreso than the unpackDatum() method. Here is the code for that.

```
bool CRawADCUnpacker::isEOE(uint32_t word)
{
  return ((word&TYPE_MASK)==TYPE_TRAIL);              ❶
}

void CRawADCUnpacker::unpackEOE(uint32_t word, ParsedADCEvent& event)
{
  if (! isEOE(word) ) {
    string errmsg = "CRawADCUnpacker::unpackEOE() ";
    errmsg += "Found non-data word when expecting data.";
    throw runtime_error(errmsg);  ❷
  }

  event.s_eventNumber = (word & TRAIL_COUNT_MASK);    ❸
}
```

❶ Check that bits 24-26 contain the number 4 to identify it as an end-of-event word.

❷ If the word is not identified as an end-of-event word, we do not understand the data properly and need to stop parsing.

❸ Using bitwise arithmetic, we extract the event number from the word and store it in the event.

Alright, now that we have the parseSingle() method defined and understood, we can go back to the parseAll() method. It is very simple and resemble the unpackData() method in that it just calls

parseSingle() until it is done. The only differences are that it stores each ADC event into a list for returning and has to protect itself from processing the BERR words (0xffffffff) that follow the more meaningful data. Here is the parseAll() implementation:

```
vector<ParsedADCEvent>
CRawADCUnpacker::parseAll(const Iter& begin,
                          const Iter& end)
{
  vector<ParsedADCEvent> parsedData;  ❶

  auto iter = begin;
  while (iter != end) {

    if (*iter != 0xffffffff) {              ❷
      auto result = parseSingle(iter,end);   ❸

      parsedData.push_back(result.second);   ❹
      iter = result.first;                   ❺
    } else {
      ++iter;                                ❻
    }
  }

  return parsedData;
}
```

❶ We initialize an empty vector that will ultimately be filled and returned.

❷ Check if the word is a BERR. We expect BERRs at the end of the event body, so seeing one or two is not an error.

❸ Parse a single event.

❹ Append the parsed data from the previous call to the end of our vector. The value returned by the parseSingle() method is actually a pair of values, the parsed data was the second element.

❺ Use the iterator returned by the parseSingle() method to increment our pointer.

❻ In the case that the word is a BERR, we need to step to the next word.

That is it for the unpacker. Let me now explain what would need to be done to make this fully independent of SpecTcl. The only dependency that this has on SpecTcl are some header files and the `TranslatorPointer<uint32_t>` type. What may not be terribly clear is that the `TranslatorPointer<uint32_t>` behaves almost identically to a `uint32_t*`. In that case, one could just as well use a `uint32_t*` pointer in its stead. You may wonder why I didn't do this already. Well, the kicker is that a TranslatorPointer is able to automatically handle byte-order swapping if the data requires it. A plain old pointer is not so smart. You will only be bitten by this if the data in the data stream was generated in such a way that swapping bytes is necessary. Only you are the one who knows this if you venture outside the world of SpecTcl. If you have to swap bytes, this becomes a bit less portable and you will need to create your own byte-swapping utility. The good news is that it is little difficulty to accomodate any solution you have concerning the TranslatorPointer. Because we hid this behind a

typedef, we only need to change the `Iter` alias to be a different type. Arguably the more general solution would be to transform the class into a template itself where the `Iter` is the template parameter. Doing so requires more changes to the code but would not have to be recompiled whenever you wanted to change the type. The most important thing here is that these changes never require a change to the actual implementation code. The only requirement is that whatever you choose to use for the iterator implements the operators used. Any random access iterator will fulfill that requirement, but you need not go so far to implement all requirements of a random access iterator. As for the included headers, you can use preprocessor conditionals for this. If you are building with SpecTcl, then you will need the proper headers, otherwise, you should not include them.

## 6.3. Setting up the Event Processing Pipeline

The `MySpecTclApp.cpp` contains the code the defines the event processing pipeline. We need to add an instance of the CRawUnpacker class to it in order for our code to execute. You will notice that in MySpecTclApp.cpp there are already a handful of predefined event processors that are registered to the pipeline. We need to make the following changes. First, find the following two line:

```
static CFixedEventUnpacker Stage1;
static CAddFirst2          Stage2;
```

You should replace the two lines with a single line:

```
static CRawUnpacker gRawStage;
```

You will also need to add the following line to the list of includes at the top of the file.

```
#include "CRawUnpacker.h"
```

The next thing you need to do is find the method called `CMySpecTclApp::CreateAnalysisPipeline()`. In it your should replace the entire body to look like:

```
void
CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
  RegisterEventProcessor(gRawStage, "Raw");
}
```

With that, SpecTcl will pass every ring item of type PHYSICS_EVENT to our event processor.

Typically, after the first event processor there might be subsequent processors that compute the values of other tree parameters from the raw value extracted by the first. A good example of this is computed a calibrated value from the raw value. It is common and recommended practice to write the subsequent event processors in such a way that they depend only on the tree parameters that were assigned values in the first processor. There are many ways to accomplish this. One method would be to separate the tree parameters from the CRawUnpacker class and define a structure that provides other processors access to them. Another method might be to define them as a static member of the CRawUnpacker class. These are just two ways that pop into my head, but you can solve the problem however best makes sense to you. Remember that SpecTcl is a software framework much like other analysis frameworks, e.g. ROOT. If you can code it in C++ and work within the constraints of the SpecTcl framework, nothing stops you from doing that.

## 6.4. Building SpecTcl

Because we have added two extra classes to the SpecTcl application, we need to ensure that they are added to the build. That is easily done by modifying the OBJECTS variable. You should edit your `Makefile` to look like this:

```
OBJECTS=MySpecTclApp.o CRawUnpacker.o CRawADCUnpacker.o
```

Furthermore, because we have made use of a handful of C++11 features, we need to ensure that the compiler operates in the C++11 mode. This is accomplished with the `-std=c++11` flag. You should add it to the `USERCXXFLAGS` variable. In the end, that line will resemble this:

```
USERCXXFLAGS= -std=c++11
```

We can now compile our program. You should see this succeed with no errors. Compilation is initiated by the **make** command.

```
spdaqXX> make
```

# Chapter 7. The VMUSBSpecTcl Alternative

Now that we have a fully implemented tailored SpecTcl, I will likely upset you by telling you that a program called VMUSBSpecTcl is available with SpecTcl distributions newer than 3.3-009. This program provides a mechanism to build a parser for your VMUSBReadout data just by reading the `daqconfig.tcl` file. This is a very limited program that can parse only a subset of the modules supported by VMUSBReadout. It is not intended to be used for parsing actual experiments. However, it is perfect for small setups. You can read more about it by pointing a browser at `/usr/opt/spectcl/3.4/share/vmusb/index.html`

```
spdaqXX> firefox /usr/opt/spectcl/3.4/share/vmusb/index.html
```

That documentation will teach you how to set up the program. However, the next section will still be useful to teach you how to create histograms, so don't stop reading here.

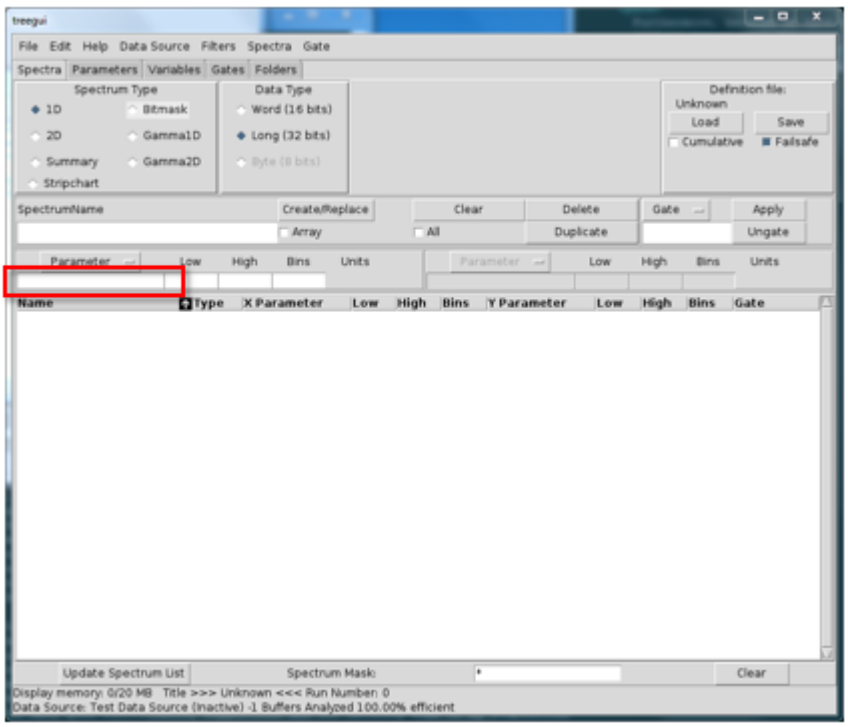
# Chapter 8. Using SpecTcl

To start using SpecTcl, you need to define at least one histogram and then attach to a data source. Let's start by launch SpecTcl. In the directory that you have copied the skeleton into, there should be a `SpecTclRC.tcl` file. This is a Tcl script that gets sourced upon start up. We will not dwell on the details of it here, but you need to make sure it is present. In that directory launch SpecTcl:

```
spdaqXX> ./SpecTcl
```

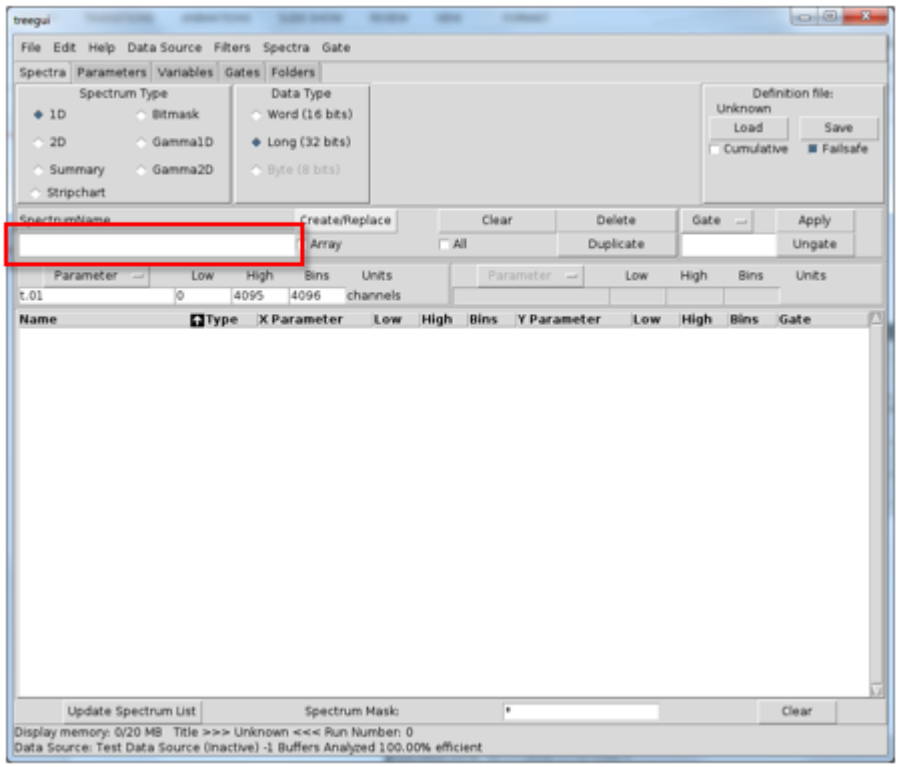
SpecTcl can take a while to start up so give it 5 to 10 seconds to complete its initialization routines. Once it is up and running, you should have four new windows open that correspond to Xamine, the TreeGui, the SpecTcl control panel, and tkcon. The definition of histograms is accomplished in the TreeGui. Without exploring all of the options for histograms, a one dimensions histogram is created by selecting a tree parameter to associate with it and also a name to refer to the histogram by. Let's create our first 1-d histogram:

1. In the middle of the window on the left, there is a text entry for a tree parameter name. We know our tree parameters are named "t.XX" so we can choose an index and write the name. Here I choose "t.01". In the following figure, I have highlighted the text entry in red.

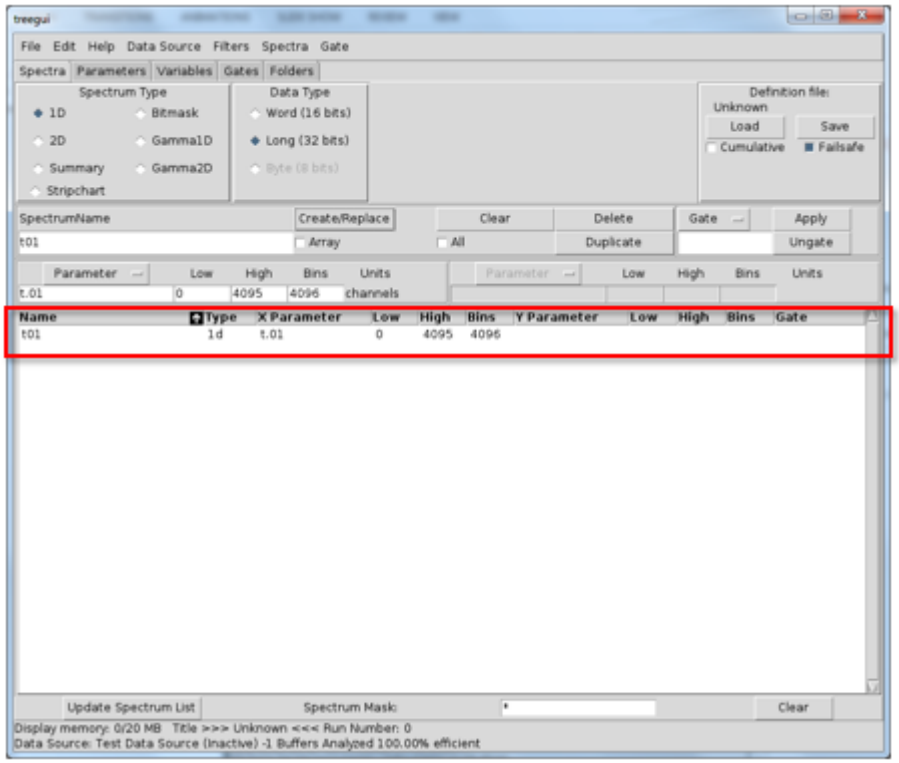



If you do not know the name of your tree parameter, you can use the "Parameter" button above the text entry to select one.

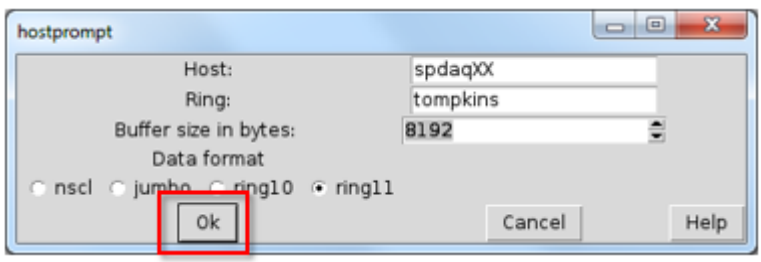
2. Next we need to give our histogram a name. Lacking any sense of creativity at the moment, I choose to name the histogram "t01". Enter this is the text entry below the label "SpectrumName". In the following figure I have hightlighted it in red.




3. To actually create the histogram, you need to press the "Create/Replace" button to the upper right of where you specified the name of the histogram. Once you have pressed this, a row should have been added to the previously empty list of histograms (once again it is highlighted in red). The window should look like this:

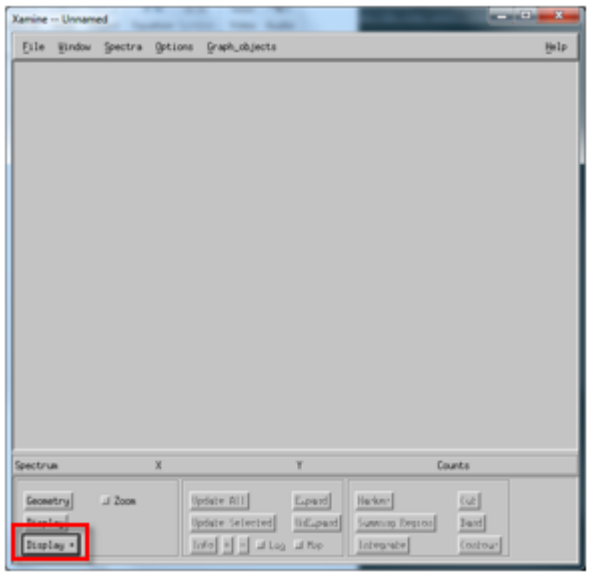



With a histogram created, we need to attach SpecTcl to a data stream. Let's assume that your Readout program is running and spewing out data. SpecTcl can be directed to attach to this "online" data stream by going to the "Data Sources" drop-down menu and selecting "Online...". A dialogue will pop up that will direct you to specify a ring buffer name. Let's assume that the ring name is "tompkins" and the ringbuffer lives on computer whose hostname is "spdaqXX". You can disregard the buffer size because it does not influence parsing when dealing with ring items. Instead, we will select that our data format is "ring11" to indicate that the data is in the 11.0 data format. At the end your dialogue should look something like this:

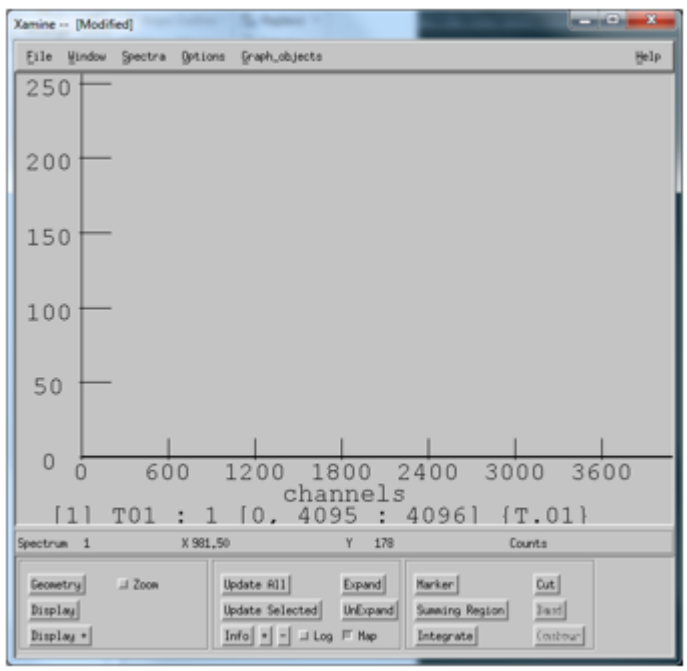



When you have the appropriate data source specified, you can press "Ok" (highlighted in red in the figure).

The status bar at the bottom of the tree gui should show that it is connect and processing events at this point. The next thing to do is move to the Xamine window and display our histogram. The bottom left corner of the Xamine window has a button labeled "Display+".




Press the "Display+" button and select a histogram to display. In my case, there is only a single histogram named "t01" to choose from. Click on the histogram name and press "Okay". The histogram should now be visible and Xamine should look something like this:



You can press "Update All" to see the histogram grow in counts from here on out. With a single histogram created, you should be able to figure out how to create other 1-d histograms and even 2-d histograms. The process is very similar. Be aware, that you need to create a histogram prior to attaching to a data source for it to receive any data.

# Chapter 9. Conclusion

The contents of the tutorial touched on the tip of the iceberg concerning how to run an experiment using NSCLDAQ. However, at this point, you should be well acquainted with the principles associated with VMUSBReadout and SpecTcl. More complicated systems simply require longer configuration files and more detailed SpecTcl event processors.