JRML DAQ Tutorial
Kevin Carnes
March, 2021

Overview

The James R. Macdonald Laboratory (JRML) uses a particular set of software packages for its multi-parameter event-driven data acquisition. This software is used for all COLTRIMS and COLTRIMS-like experiments, some VMI experiments, and time-of-flight (TOF) experiments. It is not used for any experiments involving a phosphor screen and camera to image the output of a multi-channel plate detector, which includes the majority of VMI experiments. The software runs on Linux computers and consists of two main components. The experiment control part is known as nscldaq, the main component of which is Readout. The data analysis and spectral display part is known as SpecTcl.

The software was written primarily by Ron Fox at the Michigan State University National Superconducting Cyclotron Laboratory over the last 30+ years. Although it is not commercial software and is free to copy and modify, it is copyrighted by Michigan State University and generally governed by the GNU General Public License. At JRML, Kevin Carnes has been primarily responsible for writing user-specific code to incorporate with the general MSU code, although many graduate students and postdocs have also made contributions.

Readout

The experiment control software, Readout, interfaces with a VME crate and digitization modules inserted into that crate, primarily from the Italian company C.A.E.N. A typical experiment uses three modules: a time-to-digital converter (TDC), C.A.E.N. model V1290N; an analog-to-digital converter (ADC), C.A.E.N. model V785N; and a scaler, either a C.A.E.N. V560 or an SIS 3820. Originally, the VME crate was connected to the computer via a VME control module, a fiber optic cable, and a PCI Express card in the computer. That system has mostly been replaced now in favor of a VMUSB VME controller from Wiener, Plein, and Baus, which connects to the computer via a USB cable and a USB-2 slot. The USB system is much faster than the old fiber-optic system, since the VME controller has local intelligence that reduces the necessary interactions with the computer. However, it means that the distance between the computer and the VME front end is limited to the length of a USB cable. In the documentation that follows, only the USB-based system will be described.

The first thing that must be defined on an online data acquisition system using Readout is a directory to store event files that are recorded during the run. Let's use an example directory. In the account kconline, there should be what is known as a Linux "soft link" called **kcgroup**. This points to the kcgroup space on the large networked disk array that JRML uses to store files. That link is produced with the command: *ln -s /common/kcgroup ~/kcgroup*. This should have been done for you before delivery of the computer, but it's easy to do by simply running the above command, replacing kcgroup with the name of your group. Then, let's say that a storage directory for event files will go in the directory **~/kcgroup/COLTRIMS**. Go to that directory and create a subdirectory with the Linux command: *mkdir stagearea_COLTRIMS*. Of course, you can call it whatever you like, but normally stagearea appears somewhere in the name. Then, return to your login directory (*cd ~*) and create another soft link with: *ln -s ~/kcgroup/COLTRIMS/stagearea_COLTRIMS ~/stagearea*

If this isn't done correctly, Readout will complain when you run *rdstart*.

Normally, the files involved in both Readout and SpecTcl have standard names. Therefore, the best way to separate code for different experiments is to use different directories. In the directory for Readout, there are at least four files. Let's take them one at a time.

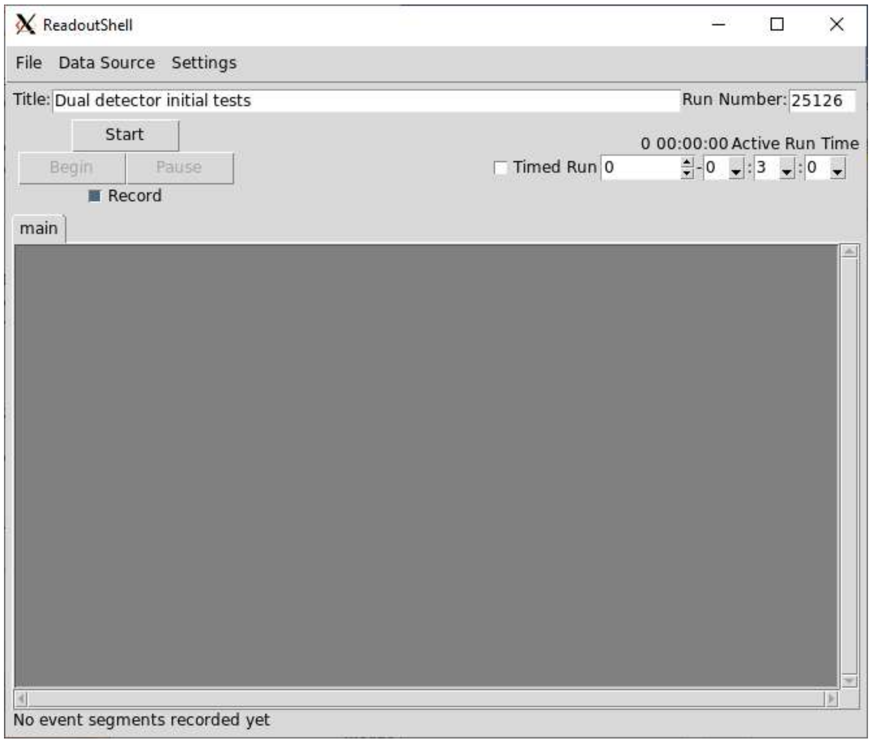
rdstart Kills any existing orphaned Readout processes and then launches the graphical run control interface. It uses environment variables to locate the code and so never needs to be modified by the user.

govmusb Contains the path to the current directory so that Readout knows where to find the configuration files. If the code in this directory is moved to another directory, this file must be modified to reflect the new path.

ctlconfig.tcl An empty file that could be used for slow control aspects of Readout, not implemented at JRML.

daqconfig.tcl This file contains all of the important run configuration information. It uses standard commands like *tdc1x90* and *adc* to modify parameters for the V1290N TDC and the V785 ADC. For most users, only the *-window* and *-offset* lines need to be modified, since they set the width of the TDC acceptance window in 25 ns steps. The *stack* command indicates what will trigger the event (almost always set as *-trigger nim1*, meaning that the I-1 input on the VM-USB VME module triggers with a negative NIM signal) and which modules are included. One other important line is the one with the *-delay* parameter. This sets the amount of time in microseconds to wait after receiving the I-1 trigger before reading the modules. The delay is especially important when the TDC common signal is also used as the I-1 trigger signal and an ADC is present. The delay parameter allows time for the ADC to finish digitizing before it's read out, since that usually comes after the TDC is ready. Usually a value of 2-4 microseconds is sufficient, but it's best to set this by trial and error for a particular experiment. When a scaler is used, additional code is necessary. If that scaler is a C.A.E.N. V560, the file **CAENV560.tcl** must be present in the directory and sourced in the **daqconfig.tcl** code. For both the V560 and the SIS3820, a scaler stack must be created with a software trigger (*-trigger scaler*) and a period in seconds (*-period*). For the V560, additional code must be included, since the driver had to be written at JRML and added manually. It's best to get these lines from existing code in other experiments.

The run control gui is fairly simple. When it first comes up, it looks like this:



Title: Dual detector initial tests
Run Number: 25126
0 00:00:00 Active Run Time
Timed Run 0 - 0 : 3 : 0
No event segments recorded yet

Initial setup is done using the *Data Source* menu. It tells Readout where it's getting its data from. *Choose Data Source -> List* to see if any sources are currently attached. If some are, click Ok and then go to *Data Source -> Delete…* Click on the top line that says *SSHPipe*, and click *Ok*. That deletes the source. Now you need to add one. Go to *Data Source -> Add…* and choose *SSHPipe*. You'll be prompted to enter the name of a Readout program. It's simplest to click on *Browse* and then select *govmusb*. Click on *Open*, then *Ok*. Now you're ready to start. This only initializes the code but doesn't actually begin the run. That will be done with the *Begin* button.

In the darkened lower part of the ReadoutShell (run control) gui, messages will pop up. There are also tabs at the top of the message area that can be clicked on to see more messages. If any of these tabs appear with red lettering and a red X, an error has occurred. If not, then you may proceed with starting the run. A few other buttons/text boxes are relevant. If you click the box next to the *Record* label, event files will be written in the stagearea directory (actually in a subdirectory called **~/stagearea/experiment/run###**, where ### is the run number). If not clicked, the experiment will run but no data will be recorded. You may type in a title for this set of runs which will be stored in the header of the event file. You may also type in a run number. Readout should automatically set this number based on the highest run number found in the stagearea event files, but sometimes you want to set it manually. Finally, you can allow a run to go for a fixed time by clicking the *Timed Run* box and entering a run duration time in the blanks provided for days, hours, minutes, and seconds.

All of these settings, along with other variables, are stored in the file **~/stagearea/.settings.tcl**, which is executed when rdstart is run. As long as nothing has changed between runs, this ensures that you're ready to go as soon as the run control interface comes up.

SpecTcl

The data analysis and spectral display package is known as SpecTcl. It is used in combination with Readout for analysis and display of live data (which we'll refer to as "online" DAQ) and separately for analysis and display of the event files recorded during an experiment (referred to as "offline" analysis). Increasingly at JRML, SpecTcl is used mainly in online mode. For offline analysis, it is often used only to reconstruct delay-line anode and MCP timing signals then to output the data in binary files for reading into other analysis programs written in MATLAB or Python. In both cases, however, a working knowledge of SpecTcl is required, hence the need for documentation such as this.

Connecting to Sorting Computers and LINUX basics

For online use, SpecTcl is run on the same DAQ system as Readout. As of this writing, these online machines are henry, foucault, avogadro, fizeau, and purcell. Although henry is usable as is, it is located in an inconvenient place and the computer ampere is usually used to control it. We also have twelve machines available for offline sorting: auger , fourier, oppy, stern, stark, zernike, strutt, wilson, siegbahn, rayleigh, roentgen, and rutherford. These are all "headless" rack PCs in CW31 that can only be accessed remotely via *Private Shell*, which should be in the start menu on your PC. When run, it begins in a *Log in – Server parameters* screen that lets you recall saved setups or create new ones. Here, you mainly want to check that the *Forward remote X11 connections to local display* box is checked in the *Tunneling* menu, accessed by clicking on the *Edit…* button. For X windows to work correctly, the *Xming* X server must be running on your machine. This should have happened automatically when you logged in, as indicated by a black X inside a red ring in the lower right system tray of your status bar at the bottom of the screen. If either *Private Shell* or *Xming* doesn't work as advertised, see PCSC.

To sort, you should log into one of the above machines as *##online*, where *##* is replaced by the initials of your group leader (*ar*, *ib*, *vk*, *cb*, or *dr* for now). All people in each group should log in using the same *##online* account and password, set by the group. Remember, you are logging in to LINUX box, so Microsoft Windows commands no longer work. For example, unlike Windows, LINUX is case sensitive. If you want to be able to open another window without logging in again, click on the *Terminal* button in the menu bar of your *Private Shell* window.

To change directories, do a

cd ~/argroup/COLTRIMS

for example. (The tilde (~) translates to the home directory of the user logged in, in this case *aronline*. Note that each group has a directory defined as **##group**, where ## are the same group initials used in the *##online* login name. All files under this **##group** directory are located on the departmental networked SANS and are backed up regularly by PCSC. Anything above the **##group** directories, say in a directory **~ibonline/mydir**, is on the local LINUX machine and not backed up.

To create a directory, use the command *mkdir*. For example, to create the directory **mydir** under Itzik's SANS group directory, use

mkdir ~/ibgroup/mydir

To copy a file to **mydir** from another directory called **olddir**, use:

cd ~/ibgroup/mydir

cp ~/ibgroup/olddir/oldfile.txt .

(That's a _space_ period at the end, important. The period means the current directory in LINUX.)

This will put the file **oldfile.txt** in **mydir**, using the same name. To copy a directory and all of its files to another directory, use the following:

cp –r ~/ibgroup/olddir/fildir ~/ibgroup/mydir

This will create the directory **~/ibgroup/mydir/fildir** containing all of the files that **~/ibgroup/olddir/fildir** contained.

A few other useful LINUX commands are:

- List the contents of a directory:
  ls (add -l to see more details, add *.xxx to see files of that form)
- Move a file
  mv location/filename newlocation/newfilename
- Remove (i.e. delete) a file
  rm location/filename.xxx
- See the full documentation for a command:
  info command (use *'Ctrl'-c* to exit)
- Get a man page (help documentation) for a command, sometimes shorter than info
  man command
- Search text files for a word
  grep word filenam (use *-i* for non-case sensitive, add *.XXX to search files of that form)
- See the contents of a file one page at a time
  more name.xxx (this will type the file on screen; press the spacebar for the next page)

The above commands allow you to manipulate files with the command line. Many prefer to do so graphically, as in Windows Explorer. Online machines will have a File Manager available from the Application menu. For offline machines, a good tool for this is the File Manager *thunar*. To run, type:

thunar &

on the command line. (Note, the ampersand, *&*, means to run the program in the background and return you to the prompt so that you can run another program.) The program will start by displaying whichever directory you're in when you run it. You can replace that default directory with any starting path that you wish, such as **~/kcgroup/VM-USB**, and it will open to display that directory.

<u>TCL (Command language) Files</u>

You will notice that you have several ***.tcl** files in your sorting directory. These are Tcl/Tk command language files. Tcl (pronounced "tickle") is an open-source scripting language that is fairly old but still widely used. Tk is the graphical user interface toolkit for Tcl. Lots of online resources and published books can help you be as productive in this language as you wish, but for now we'll just focus on the SpecTcl-specific parts. One important Tcl file is usually called **setup.tcl**. This is where your parameters and spectra are defined and other setup tasks performed. SpecTcl must have a parameter defined for every entity you want to histogram. A spectrum is then associated with that parameter. If you have a 2D spectrum, you must define two parameters, one for the x axis and one for the y. As an example, consider the following two lines from a **setup.tcl** file.

parameter adc8 108 12

spectrum adc8 1 adc8 12

The first line defines a parameter called *adc8*. Each parameter must be assigned a number, and this one is assigned number 108. This number will be used inside the sorting code to actually increment the parameter. This parameter is set up to hold numbers up to 12 bits in length, or 4096, i.e. a 4096 channel spectrum. The next line defines a spectrum that displays the histogram of the variable. The first *adc8* is the name of the spectrum (it's fine to use the same name for both parameter and spectrum). The 1 means it's a 1D spectrum. The second *adc8* is the parameter that the spectrum is histogramming, and the *12* means it, too, is 4096 channels. It is also possible to define parameters and spectra with real ranges, such as:

parameter rtof4 223 "ns"

spectrum rtof4 1 rtof4 {{0. 3000. 1000}}

Here, parameter *223*, called *rtof4*, will be histogrammed by a spectrum with a real range, and so it doesn't need a range value. The text *"ns"* will show up as a label at the bottom of the spectrum. The spectrum range definition is more complicated: It runs from channel 0. to channel 3000. and has 1000 bins. Note the double braces, required for the definition.

2D spectra can also be defined, for example:

parameter pipicox 240

parameter pipicoy 241

spectrum pipico 2 {pipicox pipicoy} {{0. 2000. 400} {0. 2000. 400}}

Note that I have to define a parameter for both the x and y variables. By comparing to the 1D definitions, this should be self-explanatory.

NOTE: the rules on names you can use are very lenient. There is no reason (other than typing) to use short parameter names or to have the same spectrum name as parameter name. For example, this would be perfectly acceptable:

parameter 2nd_recoil_time_of_flight 240

parameter 1st_recoil_time_of_flight 241

spectrum tof_coincidence 2 {2nd_recoil_time_of_flight 1st_recoil_time_of_flight) {{0. 2000. 400} {0. 2000. 400}}

It is also fine to use the same parameter for several spectra.

The spectra defined in **setup.tcl** are not automatically available for plotting. The command

sbind spectrumname

will assign a display slot in the display program (called Xamine) to the new spectrum named *spectrumname*. You can also define a series of spectra and then issue one

sbind –all

command at the end. This is typically what's done in **setup.tcl**.

***.tcl** files can also be used to set other variables that can be used inside the sorting code. This is as an alternative to using constants in the sorting code itself and rebuilding the code each time a change is made. If only a few parameters need to be set, they can be incorporated into the **setup.tcl** file. If there are numerous parameters, it is often more convenient to create a separate parameter file and read it in before sorting. (How this is done will be described below).

C++ Tips

All of the sorting code itself is written in C++. In many ways, C++ is very similar to the C language, but with some extensions, the biggest being the concept of Object Oriented (OO) programming: classes, hierarchies, inheritance, etc. Fortunately, you don't need to understand much about OO to modify and even write successful sorting code. Here are a few tips (in no particular order) to remember about C++:

- No indentation is required, but it is often used to make code sections and iterative loops more readable.
- Comments can either be as in C, where */** starts a comment and **/* ends it, no matter how many lines are in between, or the C++ specific *//*, meaning everything following on the same line until the carriage return is a comment.
- Each line of a C or C++ program (with some exceptions) must end in a semicolon, " *;* ".
- Header files, basically chunks of code that get added to the source code and often have the extension .h, are included with an *#include* command. If the command is in the form *#include "filename.h"*, with the header name in quotations, the compiler (actually the "preprocessor") looks for the file in the local directory. If it is written *#include <filename.h>,* it looks in one of the standard include directories that the compiler knows about. (This can be changed in the Makefile).
- There is no implicit variable typing. All variables must be explicitly defined, using, for example, *int* for integer and *float* for real. Other modifiers can be used in front of these, such as *constant* or *unsigned*. The author of SpecTcl has defined shorter names for many of the combined types for use in the program, such as *UInt_t* for *unsigned int*.
- Arrays use square brackets [], not parentheses (), as in *evarr[256]*. Typically, the array index starts at 0, so that a 256 element array runs from 0 to 255. Two dimensional arrays use multiple brackets, as in *rec[3][16]*.
- Output can be done easily by using the operators *<<* and *>>* and names for standard error and standard output, *cerr* and *cout*. For example, the line *cerr << "Too many hits on channel " << chan << endl;* replaces the variable *chan* with its value and writes the line to standard error, which will appear on your terminal screen. The variable *endl* has been defined to mean end-of-line and line feed.
- Variables can be incremented or decremented by 1 with the *++* or *--* operators, as in *count++;* to increment the variable count.

- Braces *{}* are used to mark off sections of code that go together, such as subroutines, *if* blocks, or loops (*for* and *while* loops).
- Logical comparisons in *if* statements use *==*, *<*, *>*, *<=*, *>=*, *!* (for "not"), *&&* (for "and"), and *||* (for "or"). One of the most common mistakes in C or C++ is to use a single = for equality comparisons. The statement *if (a = 1)* actually sets the variable a equal to 1 instead of giving a logical value if the two are equal (the proper form is *if (a == 1)*).
- Pointers are often a difficult concept to grasp. A pointer is a variable whose value is the address of another variable, not its value. So, assume we have an integer variable *ival*. The pointer to *ival* would be declared with *int* pval;* meaning that *pval* points to an integer. Then, *pval* can be assigned with *pval=&ival;* where the *&* operator means "address of". If *pval* is incremented, with *pval++*, it means that it now points to the next address location. If instead the form **pval* is used, as in *(*pval)++*, the content (*ival*)of the address pointed to by *pval* is incremented by one. Obviously, proper use of parentheses is important here. This is especially useful when passing variables as arguments to subroutines. By default, these variables show up in the subroutines as copies of the originals, so changes made in the subroutine to the copies are not made to the originals once the program returns from the subroutine. However, if pointers are passed as arguments, the subroutine can make changes to the original by using the **pval* form, since the copy of the address still works as the address of the original variable.
- Instead of the *do* loop in Fortran, C and C++ use a *for* loop. The format is

```
for(int i = 0;i < imax;i++) {
.
.
}
```

  The variable *i* is only defined for the duration of the loop, so it must be defined as an *int* and initialized. The loop starts with *i = 0* and stops when *i* is greater than or equal to *imax* (the iteration for *i=imax* is not executed). *i* is incremented by 1 for each iteration. The braces aren't necessary if there is only a single statement in the *if* block. A similar loop is the *while* loop, which continues to execute the loop as long as the condition in the *while* statement is true. For example,

```
while (*pl != 0x0000C0FF) {              // Execute as long as the value at pointer
// location pl is not equal to Hexadecimal 0000C0FF.
  evarr[evCount] = *pl++;          // Copy the value of the variable pointed to by pl
// into array evarr, then increment pointer to point to next item in buffer.
  evCount++;                       // Increment event counter.
 }
```

- C and C++ don't have as many built in math operators as Fortran. In particular, there is no operator for taking a number to a power, such as *** or *^*. If a variable just needs to be squared, it's easiest to write it out as a multiplication (*ivar * ivar*). For larger powers, library routines can be used (*pow(d,e)*, *d* to the power *e*).

- The OO idea of classes does directly affect how "subroutines" are defined in C++. A class is typically first declared in a header file with the *class* keyword, and all subroutines that are part of that class are declared within the braces *{}* of the class. For example,

```
class CLVEventDecoder : public CEventProcessor
{
```

```
void Resort(int arg1, int arg2);
}
```

defines the class *CLVEventDecoder*, which is based on the class *CEventProcessor*, and declares one of its routines *Resort*. Then, when the routines are actually defined, usually in the *.cpp file, they are referred to by the form: *classname::routine*, such as *CLVEventDecoder::Resort(int arg1, int arg2)*. This means that the routine *Resort* is part of the class *CLVEventDecoder*.

- Each class has "private" data that can only be accessed by subroutines in the class and that survives between subroutine calls. The data variables are defined in the class definition in the header file and typically initialized in a class subroutine called the constructor (just a subroutine with the same name as the class). The initialization can be done with a *(0)*, such as

```
CLVEventDecoder::CLVEventDecoder() :
  m_nEventCount(0),
  m_nGoodCount(0)
{
}
```

  where the private data variables *m_nEventCount* and *m_nGoodCount* are both initialized to 0.

Remember, C++ is a very powerful language, and it will take a lot of work to master it. However, these few tips will hopefully answer many of the questions that come to mind when looking at C++ source code for the first time.

SpecTcl Sorting Code Logic

SpecTcl sorting code must conform to the structure that SpecTcl requires. The first step in understanding that structure is to look at the sorting class subroutines that have special meaning. If the class name is *CLVEventDecoder*, for example, these routines are *CLVEventDecoder::OnAttach*, *CLVEventDecoder::OnBegin*, *CLVEventDecoder::OnEnd*, and *CLVEventDecoder::operator()*. Commands included in the *OnAttach* routine are run when SpecTcl starts up for the first time. *OnBegin* commands are run at the beginning of each run, assuming that the data buffer has a "begin" event (usually automatically inserted into the event file by the recorder). *OnEnd* is after an "end run" event. These should be relatively self-explanatory, but the *operator()* routine is perhaps non-intuitive. C++ allows something called operator overloading, which is basically re-defining a standard operator to mean something else. In this case, the parentheses operator *()*, when used with the *CLVEventDecoder* class, is defined to carry out the commands in the *operator()* code (between the *{* and *}*). The actual call of this overloaded operator is something you will never see, as it is buried deep in the internal code of SpecTcl. However, you only need to know that what happens in the *operator()* routine is the code that gets executed for each event. This is where the bulk of the sorting actually takes place. Other subroutines can be defined and called by the user from the basic SpecTcl routines, usually operator().

SpecTcl parameters are histogrammed in two ways, depending on if the parameter is to be incremented only once per event or multiple times per event. The first case is much easier, only requiring a line such as:

```
rEvent[208] = xr2;
```

which increments parameter 208 at position *xr2* by 1. (Remember, the parameter is what is processed; the spectra are histograms of the parameter). This is different from other systems like ROOT where a histogram increment is explicitly done. It also means that there is no simple mechanism to do multiple increments on a parameter in a single event, such as incrementing a time-of-flight spectrum parameter with all time hits. This requires manipulating the spectrum itself and involves a lot of code overhead. It looks daunting, but the basic format can be used for all multiple-increment spectra. Let's consider a multiple TOF spectrum. The first step is to make the spectrum part of the private class data with the line:

CSpectrum m_prtofall;*

in the class definition in the header file. *CSpectrum** means that *m_prtofall* is a pointer to the class *CSpectrum*. (Note, as with most variables, the form of the pointer variable is purely convention. It can take any form.) The pointer is initialized to 0 in the event decoder constructor just like any other variable, with *m_prtofall(0)*. Next, the actual spectrum has to be found for m_prtofall to point to.

```
string strrtofall = "rtofall";          // Define a string variable to hold the spectrum name
m_prtofall = pHistogrammer->FindSpectrum(strrtofall);  // Use the FindSpectrum
// subroutine in the class pointed to by pHistogrammer to find the spectrum named
// rtofall.  If found, m_prtofall will now point to the spectrum.
if(m_prtofall){                          // Make sure it found the spectrum, i.e. m_prtofall isn't 0.
 if(m_prtofall->getSpectrumType() != ke1D) {      // Make sure  it's a 1D spectrum.
  cerr << "Found "<<strrtofall<< " but it's not 1-D\n";       // If not print error and
  m_prtofall = (CSpectrum*)kpNULL;                            // reset pointer to 0 (Null).
 }
}
```

The same technique works for 2D spectra with minor changes to account for the second dimension. I have created functions called MultiPlot1d and MultiPlot2d to do the actual multiple increments, and they use the spectrum pointer and channel to increment as arguments. Note, however, that this method of multi-incrementing essentially bypasses some of the benefits of SpecTcl, such as being able to use real numbers (including negative numbers) on the axes of spectra, and having access to the built-in gating commands. Other methods are available to do this within the SpecTcl context. Basically, they involve creating separate parameters for each hit. See me for further information.

Often, variables that are seldom changed, such as physical constants or the size of a channel plate, are initialized in a header file with the word *Constants* somewhere in the name, such as C1290Constants.h. Should changes be necessary, it's easier to locate the variables when collected in one place, change them, then rebuild the code. It is also possible to initialize some of these variables in a Tcl file so that the sorting code doesn't need to be rebuilt each time a change is made to a "constant". Please see me for information on how to do this, as it is beyond the scope of this description.

Building and Running the Sorting Code

Now, on to compiling, linking, and running the sorting code. In what follows, I'm assuming your current directory is the same as the one where your sorting code resides, for example,

~/ibgroup/Xlasersort/SpecTcl. Once inside a sorting directory, you make sure that the executable file is up-to-date via the commands (in one of your regular Linux terminal windows)

make clean
make

The first command deletes all of the old compiled and linked files and is not strictly necessary if the make dependencies have been properly configured. However, it's safest to use it initially until you become more comfortable with the procedure. The second command recompiles and links the code. Be sure to pay attention to any errors that appear. If successful, the result is that you will have a file called **SpecTcl** that is the executable used to run the program. By the way, a nice feature of the shell used on our LINUX box is command completion. If you type enough of a command or filename to clearly identify it from other possibilities, hitting the Tab key will complete the command/name for you. The rules used by the make command are found in a file called **Makefile** in the directory you are in. This is a very powerful file that makes compiling and linking up-to-date code much simpler, but learning how to modify the file will take some time. Usually, you shouldn't need to touch it.

To run the sorting code for the first time, from your sorting directory, type:

./SpecTcl

The *./* means look for the SpecTcl program in the current directory. You should see at least four X objects pop up on your screen (some of them may be hidden, so look below on your taskbar in the X section to bring them forward if necessary). These are the *treegui* window, the *tkcon* window, the *Xamine* window, and the *SpecTcl* command buttons window. Note that *tkcon* is an obsolete label and the window in question just shows up now with the same SpecTcl label as the command buttons window. It is the window that has a command line interface with the current directory and a % as the prompt. I'll continue to refer to it as *tkcon* to distinguish it from the SpecTcl buttons window. The *Xamine* window is where you will view and manipulate spectra, the *tkcon* window is where all commands will be typed, the *treegui* window allows you to get spectrum and parameter information and execute some commands via a menu (rarely used at JRML), and the *SpecTcl* command buttons are for frequently used commands. Next, we'll list several steps and possible commands.

- For most groups, the next step is to define the parameters and spectra used in the sort. In the *tkcon* window, type:
  source setup.tcl
  assuming that *setup.tcl* is the name of your file that contains the definitions.
- For some groups, a separate parameter file needs to be sourced before (or after) setup.tcl. This contains definitions and sets values for variables that the sorting code will use and that don't require recompilation of the code. For example, this could be done with
  source rerun1000.tcl
- The next step depends on whether you are running SpecTcl online with the *Readout* program or offline for analysis of event files. For online use, you need to click the *Attach online* button on the SpecTcl button command menu window. A hostprompt window will pop up listing the name of the host DAQ machine you want to attach to, the name of the ring, etc. These can all be left as the default values, so just click on *Ok*. SpecTcl will now operate on the data stream coming in from the Readout code. If you wish to change something in Readout without logging out of

SpecTcl, simply click on the *Detach* button on the SpecTcl menu button window. Then, after restarting Readout, use the *Attach online* button again.

- For sorting event files, you first need to attach an event file to let SpecTcl know which file to sort. This can be done using the graphical SpecTcl menu or via the *attach* command in the tkcon window. For the menu option, click on *Attach to file*. A selection window will pop up. You can navigate by typing the directory holding the event file (in the **experiment** subdirectory of your **stagearea** directory) directly into the *Filter* line, or by clicking on the .. line in the *Directories* list to go up a level and then back down to the **experiment** directory. You eventually have to click on the run directory you want, for example *run25114*, and then ***.evt** files will show up in the *Files* pane. Finally, click on the event file you wish to sort and then click the *Ok* button. The *Buffer size* and the event file type (*ring11*) should be left at their default values. If the event file you clicked on is part of a "segmented run", i.e. a long run whose event files were automatically written in 2 GB segments with names like **run-25114-00.evt**, **run-25114-01.evt**, etc., a *Multi file run* window will pop up asking if you want to play back the entire run. If you click *Yes*, all files will be sorted. If you click *No*, only the file you selected will be sorted. Once finished selecting the file and how to sort it, the sorting will begin immediately, so make sure you've already sourced any parameter file that you wish to apply to this run.

- Rather than go through the multiple mouse clicks required to attach an event file using the graphical menu, many choose to embed the *attach* command in a ***.tcl** file, typically also containing commands to set parameter values, which can then be sourced using the *tkcon* window. A standard command would be, for example:

  attach -format ring -file /common/kcgroup/VM-USB/stagearea/experiment/run109/run-0109-00.evt

  Note that the default value for size is used. Also, the sorting has to be manually started when the attach command is used by clicking on the *Start Analysis* button on the SpecTcl menu.

  To combine several data files into one sort, the following will work:

  *attach -format ring -pipe cat /common/ kcgroup/VM-USB/stagearea/experiment/run109/run-0109-00.evt *
  */common/ kcgroup/VM-USB/stagearea/experiment/run109/run-0109-01.evt *
  /common/ kcgroup/VM-USB/stagearea/experiment/run109/run-0109-02.evt

  This takes advantage of a *pipe* in Linux that hooks up the output of one command, *cat* in this case, to the input of another, here the *attach* command.

- To Start/Stop sorting, click on the *Start/Stop Analysis* button on the SpecTcl command window. Spectra can be cleared with one of those buttons, and 1D and 2D spectra can be exported to a text file for reading in to Origin.

- To look at spectra in the Xamine window, you first have to define a set of display panels and choose which spectra to display. Predefined sets of spectra may already be defined These can be accessed by the *Window->Read Configuration* buttons on the Xamine top menu bar. The various window sets are displayed under the *Files* heading. Select one and click *OK*. If you want to define a new window set, click on the *Geometry* button at the lower left of the Xamine screen. This allows you to configure the display for up to a 10x10 matrix of spectra. After selecting your display geometry, you have to select specific spectra to go in each slot. The

Display button allows you to select a single spectrum for a single slot. It produces a scrolled list of all the spectrum names you have defined. Selecting one with the mouse (or typing the name in the text box) and clicking *OK* will display that spectrum in the display slot. The *Display+* button will automatically step through each available slot as you select a spectrum and hit the *Apply* button. Use the *OK* button for the last slot you want to fill. If you want to save this particular display configuration, the *Window->Write Configuration* buttons will let you give it a name. It will automatically be given a **.win** extension.

- A given spectrum can be chosen to fill the display by double clicking on it. Double clicking again will reduce it to its former size. (The *Zoom* button does the same thing.) Part of a spectrum can be expanded using the *Expand* button. A window will pop up to let you enter the coordinates at each end of the area you want to expand, or you can click on them with the mouse. For a 2D spectrum, the coordinates are at opposite corners of a rectangular region that you want to expand. Note that sometimes this window pops up behind the Xamine window and can be hard to see. You can look on the taskbar at the bottom of your PC in the *X # Xming* group to find the Expand window and click on its name to bring it to the front if necessary. The *UnExpand* button removes the expansion.
- To calculate the area of a peak in a spectrum, first define a *Summing Region* with the Xamine button of the same name, then click the *Integrate* button. These are simple integrations, i.e. a sum of the counts in each channel between the summing region boundaries. The centroid and FWHM are also calculated. For background subtraction or fitting, the data must be exported to Origin. Summing regions can be removed by clicking on the *Graph_objects* pull down menu at the top of the Xamine screen and selecting *Delete*. You'll get a list of graphical objects defined for that spectrum, including summing regions, and these can be selected to delete.
- To overlay one spectrum with another, click on the *Spectra* pull down menu at the top of Xamine and then click on *Superimpose*. To remove an overlay, click on *UnSuperimpose*.
- The *Log* button in the middle of the bottom part of the Xamine screen toggles back and forth between a log scale on the Y axis for 1D spectra or the Z axis for 2D spectra.
- The *Map* button next to the *Log* button toggles back and forth between displaying the channels as integers, 0 to the maximum channel in that dimension, or as the spectrum was defined, from the defined minimum to maximum channel numbers. If the spectrum was defined as a number of bits in size, these two will be the same. Note that this feature doesn't work if the spectra were defined using the MultiPlot subroutines, only for the rEvent method.
- The *Marker* button puts a dot on the spectrum. The *Cut* button sets a 1D gate, meaning any event with points between the gate limits satisfies the gate. *Band* sets a 2D polyline, meaning anything below the line on a 2D spectrum satisfies the gate. *Contour* sets a closed shape, satisfied when an event has points within the shape. Read the more extensive online SpecTcl documentation for how to use gate commands.
- Other options can be accessed from the menu buttons at the top of the Xamine screen, such as setting Spectra properties like *Autoscale*. You can experiment with these.
- To print a spectrum, click on the *File* menu and select *Print...* Usually, I select *Landscape*, *Print Selected Spectrum*, and *Specify size in Spectrum Options*. Then, in the *Spectrum Options* tab, I set the size to be 9" wide by 6" high. To print all of the spectra on a multi-spectra display, select *Print All Spectra* and choose the number of *Rows* and *Columns* to match the display. Output options can be either *To File* or *To Printer*. If Printer, the *Print Command* has to have the right printer name after the *lpr –P*. To find which printers are available, execute the following command from a terminal:

  lpstat -p

Note, the color scheme won't match the scheme on the display, nor will the printout be publication ready. For more control over printing, it's best to export a spectrum to Origin.

- Spectra can be saved and then read back in with the *swrite* and *sread* commands. The format of these commands can be found in the more complete online SpecTcl documentation. These are best for saving and reading a single spectrum. Although several spectra can be written to a single file with *swrite*, *sread* will only read the first spectrum in that file unless it is called from within a more complex Tcl procedure. Remember that any existing spectra will be overwritten. To export spectra as ASCII files that can readily be read into Origin for processing, use one of the two buttons at the bottom of the SpecTcl commands window menu. *Write Spectrum (Real)* will write out the axes using the real values defined in setup.tcl spectra definitions. For example, if a 1-d spectrum is defined as {{-40. 40. 801}}, two 801-row columns will be output. The first will be the spectrum coordinate ranging from -40. to 40., and the second will be the counts in each index. If *Write Spectrum (Int)* is selected, the only difference in output is that the first column will be from 0 to 801, not -40. to 40., i.e. the integer indices of the spectrum.
- To make sure the spectra are updated in the Xamine display during the sort, choose *Options* and select *Update Rate*. Set the slide bar to the number of seconds you want between updates and click *Apply to All*.
- SpecTcl has a command for taking projections of 2D spectra. Its format is:

  project [-[no]snapshot] sourcespec newspec x|y [contourname]

  where you choose "x" or "y" depending on which direction you wish to project. The *-snapshot* option makes a projection at a single instant in time. *-nosnapshot* creates a spectrum that is incremented when the sort is continued, or even cleared and restarted. Here, *sourcespec* is an existing 2D spectrum and *newspec* is the name of a new, nonexistent 1D spectrum. Instead of manual channel limits, an optional *contourname* can be given to limit the number of channels included in the projection. This refers to a contour that has already been set. If you have forgotten the name of a contour, you can click on the *Graph_objects* pull down menu at the top of the Xamine screen and select *Copy Object*, which will give a list of graphical objects defined for that spectrum. If a *contourname* is omitted in the *project* command, all channels along one axis of the spectrum will be summed for the projection. Before displaying the newly created projection spectrum, you have to issue the command *sbind –all* in the tkcon window to let Xamine know that the spectrum exists.

Help and Additional Capabilities

Aside from this document, help for the entire DAQ system as implemented at the NSCL at MSU can be found at the following URL:

http://docs.nscl.msu.edu/daq/index.php

This was written mainly by Ron Fox and is somewhat haphazardly maintained. However, all of the built-in SpecTcl commands listed in this document are explained there in greater detail.

Clearly, Xamine is not perfect as a spectrum display program. In particular, the spectra aren't publication quality. However, the author of the code is continually upgrading it and seems responsive to suggestions. It is also a very easy thing to export spectra and read them into Origin where they can be prepared for publication (see above). Also, event files can be exported in a format that can be read in by

MATLAB, Python, or the CERN program ROOT. All three of these are powerful packages that have numerous built-in analysis tools as well as superior graphics.

For information on these additional capabilities, or if you have further questions, please contact Kevin Carnes, kdc@phys.ksu.edu.