Using NCSL DAQ Software to Readout a CAEN V785 Peak-Sensing ADC

Timothy Hoagland

Ron Fox

Using NCSL DAQ Software to Readout a CAEN V785 Peak-Sensing ADC

by Timothy Hoagland

by Ron Fox

Revision History

Revision 1.0 December 2, 2004 Revised by: TH
Original Release
Revision 1.1 March 28, 2006 Revised by: RF
Translated to Docbook

Table of Contents

List of Figures

Preface

This paper's purpose is to guide the reader in setting up and reading out data from a CAEN V785 peak sensing ADC. It covers the electronics that will be needed and how to set them up. It will show how to modify the production readout (V8.1), and SpecTcl (V3.1) software skeletons to readout and histogram data from the adc. Testing for the code is also covered to a limited extent.

This paper assumes that you are

- Somewhat familiar with Linux.
- Have a basic familiarity with the C++ programming language
- Familiar with using an oscilloscope

Final versions of the software are available for download at:
http://docs.nscl.msu.edu/daq/samples/CAENV785/CAENV785.zip

Every effort has been made to ensure the accuracy of this document. As authors we take very seriously reports of mistakes, omissions, and unclear documentation. If you come across a part of the document you think is wrong, needs expansion, or is not clearly worded, please report this to <fox@nscl.msu.edu>. We will correct this problem as soon as possible and, if you like, credit you publicly in the document for finding and helping us fix this issue.

Chapter 1. The electronics

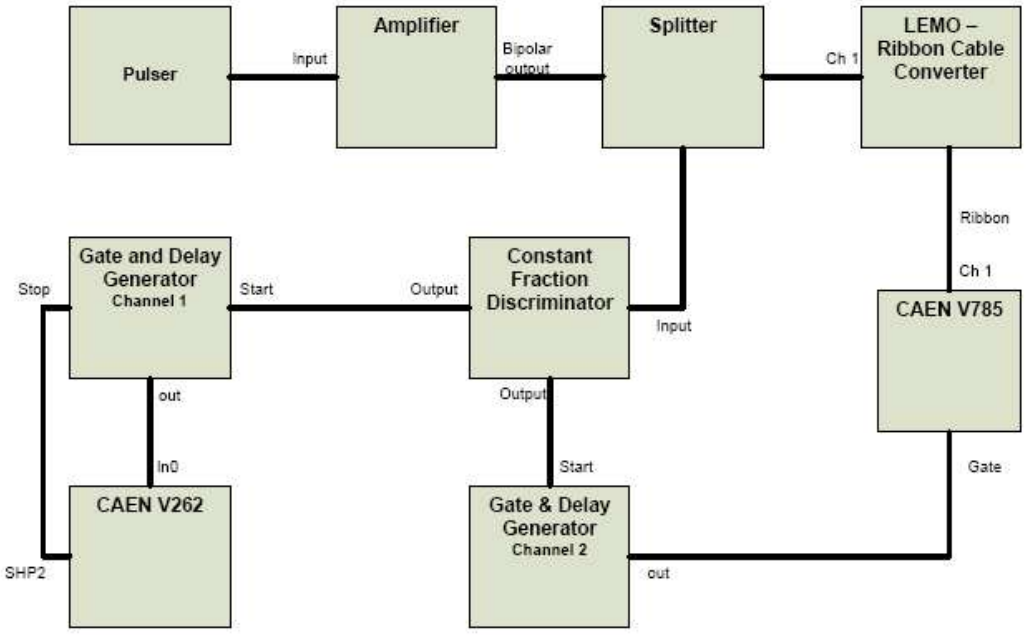
The CAEN V785 is a 12 bit, 32-channel analog to digital converter that returns a value related to the maximum voltage that occurs during the time a gate is present. The input signal must be positive and less than 4V. The V785 is a VME based module. For a complete understanding of the module I suggest that you obtain a copy of the product manual, available as a PDF at http://www.caen.it/

1.1. A minimal Electronics Setup

The following items will be needed to build our simple setup:

- CAEN V785 ADC module
- VME Crate
- SBS Bit3 PCI/VME bus interface
- NIM Crate
- NIM Spectroscopy amplifier of some sort
- NIM Discriminator
- Two channels of NIM Gate and delay generator
- NIM Signal splitter
- LEMO to Ribbon cable converter
- VME CAEN V262 I/O module
- Pulser
- 50ohm LEMO terminator
- An assortment of LEMO Cables
- A 34 conductor ribbon cable with 3M connectors on each end
- An oscilloscope

Figure 1-1. A simple electronics setup for the CAEN V785



Before starting be sure that you can secure all of these items, many are available through the NSCL electronics pool. Figure 1-1 shows the complete setup of the system, due to the varying amount of familiarity of readers to the electronics, a brief description of the component and what the signal coming out of it should look like will be given.

We will first look at the signal from the pulser. A pocket will work well for this. A pocket pulsers will give a small negative pulse, but we will invert it later. The signal directly from the pulser will look like figure 1-2 when viewed on a scope. The pulser only works when terminated with 50 Ohms.

The pulser will run directly into a NIM based amplifier. This amplifer serves two purposes. First it will amplify our signal helping it stand out against any noise we might have in the channel. It will also invert the negative pulser signal giving us the positive signal that is required by the V785. The signal coming from the bi-polar output of the amplifier will look like figure 1-3. Adjust the gain on the amplifier until you output signal has an amplitude of about 2V.

Figure 1-2. Pulser Output Signal

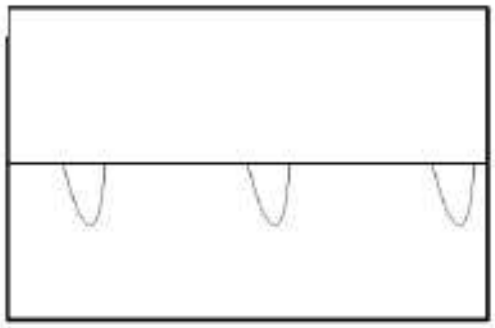
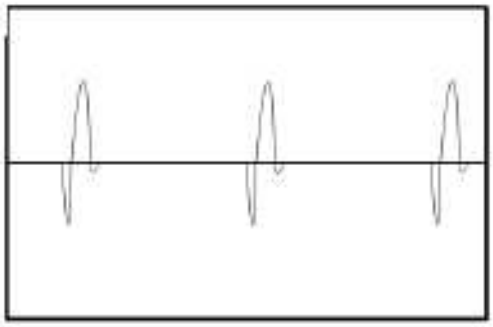


Figure 1-3. Amplifier Output SIgnal



After you are happy with the signal from the amplifier plug the bipolar output into the splitter. The splitter will simply split the signal much like a "T" but will keep the 50Ω termination needed by the pulser.

One of the splitter outputs will go into a LEMO to Ribbon Cable con- verter. This can be either a NIM based module or a freestanding adapter. Either way the function is simple and will allow you to put the signal onto a flat ribbon cable to be plugged into the CAEN V785.

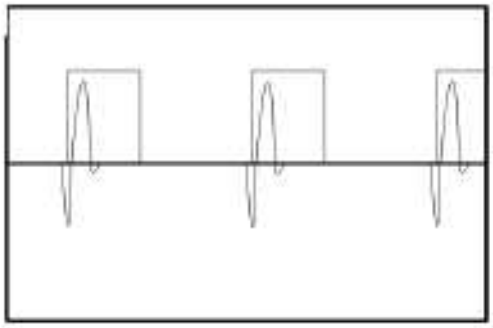
> **Note:** Ribbon cable can difficult to work with because it is easy to get it twisted and lose track, of which end is which. To avoid this look carefully at the coloring on the cable you are using and be sure it is plugged in the correct channel of the ADC.

The other output from the splitter will go into a discriminator. This could be either a leading edge or a constant fraction discriminator. The job of the discriminator is determine when a signal has occurred and to put out a logic signal when it does. The discriminator will have a threshold knob, be sure that the threshold is set above any background noise that may be present so that the discriminator only trips on the pulser signal. It will be useful to look at the both the logic pulse and the signal from the amplifier at the same time when setting the threshold value. Figure 1-4 shows what you will see.

Figure 1-4. Amplified Signal with Logic pulse

When the threshold is set to a reasonable level, connect one of the dis- criminator outputs to the start of one channel of your gate and delay gen- erator. The gate and delay generator will generate a gate when it receives a logic signal from the discriminator. When looking at the signal from the amplifier and the gate at the same time on the scope, the signal pulse should fall within the gate as seen in figure 1-5. If the gate is occurring too early you can use the gate and delay generator to delay the gate or to make the gate last longer. If the gate is too late you need to delay the signal pulse, that can be done using a delay module, or by adding cable delay. When you are happy with the gate timing, plug it into one of the LEMO connectors on the CAEN V785 labeled gate, put a 50 Ohm terminator in the other one.

Figure 1-5. Amplified Signal and Gate



The second output of the discriminator will go to a second gate and delay channel; this combined with the CAEN V262 will trigger the computer when there is data on the V785. This channel will be run in latched mode meaning that it will be given both a start and a stop signal for the gate. The output signal will start being "true" when the start signal is received and stops being "true" when the stop signal is received. The start signal is the signal from the discriminator. The discriminator output will go to the IN0 of the CAEN V262 I/O module. The stop signal is generated by the computer to indicate it has responded to the trigger. That signal is present at the SHP2 output of the V262 and should be connected to the stop on the gate and delay generator.

At this point your setup should be complete. Now is good time to make sure that your setup is the same as Figure 1-1. If everything is setup correctly the BUSY and DRDY lights on the V785 should be lit up.

> **Note:** The setup shown in figure 1 does not have a dead time lockout, that is it could try to process a second event while the computer is still busy. This will not be a problem as long as source is a predictable as a pulser but would be problematic if we replaced the pulser with a detector signal.

Chapter 2. Setting up the software

The software modifications needed to make the above setup work can be divided into three tasks:

1. Telling the software what electronics we are using.
2. Telling the software how to respond to event triggers.
3. Telling the software how to analyze the data acquired.

2.1. Readout software

In order to tell the software about our electronics we are going to develop a C++ class for our module. That class will be a derived class but we don't need to concern ourselves with the details of the parent class. Like all of the software tailoring we need to do, most of the details are hidden and we need only to fill in a few holes.

The following commands will make a new directory and copy the skeleton files to it:

```
mkdir -p ~/experiment/readout
cd ~/experiment/readout
cp /usr/opt/daq/pReadoutSkeleton/* .
```

2.1.1. Modifying the Readout Skeleton

Now that we have obtained a copy of the Skeleton file we can began to think about the modifications we need to make. We need to tell the software what kind of module we are using, how to initialize it, how to clear it, and how to read it. We will do this by creating a class called by `MyEventSegment`. In order to follow good coding practice and to make our code as versatile as possible we will write our class in two separate files, a header file and an implementation file. Start by creating a file called `MyEventSement.h`. It should look like this:

Example 2-1. Header for `MyEventSegment`

```
/*
This is the header file to define the MyEventSegment class, which
is derived from CEventSegment.  This class can be used to read
out any number of CAEN modules covered by the CAENcard class.
Those cards include the V785, V775, and  V792.
```

```
Tim Hoagland
11/3/04
s04.thoagland@wittenberg.edu

*/


#ifndef __MYEVENTSEGMENT_H                // ❶
#define __MTEVENTSEGMENT_H
#ifdef HAVE_STD_NAMESPACE
using namespace std;                      // ❷
#endif
#include <spectrodaq.h>
#include <CEventSegment.h>
#include <CDocumentedPacket.h>
#include <CAENcard.h>
#define CAENTIMEOUT 50


// Declares a class derived from CEventSegment
class MyEventSegment : public CEventSegment   // ❸
{
private:
  CDocumentedPacket m_MyPacket;             // ❹
  CAENcard* module;                         // ❺
public:
  MyEventSegment(short slot,unsigned short Id); // ❻
              // Defines packet info
  ~MyEventSegment();                 // ❼

  virtual void Initialize();                    // ❽
            // One time Module setup

  virtual void Clear();                         // ❾
            // Resets data buffer

  virtual unsigned int MaxSize();               // (10)

  virtual DAQWordBufferPtr& Read(DAQWordBufferPtr& rBuf);  // (11)
            // Reads data buffer
};
#endif
```

The header defines the class, its internal data and the services it exports to the readout framework. Refer to the circled numbers in the listing above when reading the following explanation.

❶ Each header should protect itself against being included more than once per compilation unit. This #ifndef directive and the subsequente #define do this. The first time the header is included,

__MYEVENTSEGMENT_H is not defined, and the #ifndef is true. This causes __MYEVENTSEGMENT_H to be defined, which prevents subsequent inclusions from doing anything.

❷ Newer compilers define many of the standard classes, functions and constants inside of the std:: namespace. The spectrodaq.h header refers to these without explicitly qualifying them with the namespace. If the std:: namespace exists, this directive asks the compiler to search it for unqualified names (e.g cin could be resolved by std::cin).

❸ The class we are defining MyEventSegment implements the services of a base class called CEventSegment. Anything that reads out a chunk of the experiment is an event segment and must be derived from the CEventSegment base class.

❹ By convention, events are segmented in to *packets*. A packet is a leading word count, followed by an word that identifies the contents of the packet followed by the packet body. Packet identifiers are maintained by Daniel Bazin, and the set of packet identifiers that have been currently assigned are described at: http://groups.nscl.msu.edu/userinfo/daq/nscltags.html If you are creating a detector system that will need its own tag contact <bazin@nscl.msu.edu> to get one assigned. The m_MyPacket is member data for the class. It is a CDocumentedPacket. Documented packets do the book-keeping associated with maintaining the packet structure as well as documenting their presence in documentation records written at the beginning of the run.

❺ The support software for the CAEN V685 ADC is itself a class: CAENcard. We will create an object of this class in order to access the module. This pointer will be used to access the object and hence the module.

❻ This line declares the *constructor* for MyEventSegment. A constructor is a function that is called to initialize the data of an object when the object is being created (constructed).

❼ This line declares the *destructor* for MyEventSegment. A destructor is a function that is called when an object of a class is being destroyed. Since we will be dynamically creating the CAENcard pointed to by module, we must declare and implement a destructor to destroy that object when objects of MyEventSegment are destroyed.

❽ The Initialize member function is called whenever the run is about to become active. If hardware or software requires initialization, it can be done here. This function will be called both when a run is begun as well as when a run is resumed.

❾ The Clear member function is called when it is appropriate to clear any data that may have been latched into the digitizers. This occurs at the beginning of a run and after each event is read out.

(10) The MaxSize member function is obsolete but must be defined and implemented for compatiblity with older software. In the past this returned the maximum number of words the event segment would add to the event. Now it is never called.

(11) The Read member function is called in response to a trigger. This member must read the part of the event that is managed by this event segment.

We will also create an implementation file: MyEventSegment.cpp. This file will implement the member functions that were defined by MyEventSegment.h above. The contents of this file are shown below:

Example 2-2. Implementation of `CMyEventSegment`

```
/*
    This software is Copyright by the Board of Trustees of Michigan
    State University (c) Copyright 2005.

    You may use this software under the terms of the GNU public license
    (GPL).  The terms of this license are described at:

     http://www.gnu.org/licenses/gpl.txt

     Author:
             Ron Fox
     NSCL
     Michigan State University
     East Lansing, MI 48824-1321
   */


/*

This is the implementation file for the MyEventSegment
class.  This class defines funtions that can be used to
readout any module covered in the CAENcard class.  These
include the V785, V775, and V792


Tim Hoagland
11/3/04
s04.thoagland@wittenberg.edu


*/



#include <config.h>
#ifdef HAVE_STD_NAMESPACE                    // ❶
using namespace std;
#endif

#include "MyEventSegment.h"                  // ❷

static char* pPacketVersion = "1.0";         // ❸
// Packet version -should be changed whenever major changes are made
// to the packet structure.


//constructor set Packet details
MyEventSegment::MyEventSegment(short slot, unsigned short Id):
  m_MyPacket(Id,
     string("My Packet"),
     string("Sample documented packet"),
```

```
     string(pPacketVersion))       // ❹
{
  module = new CAENcard(slot);            // ❺

}
// Destructor:
MyEventSegment::~MyEventSegment()
{
  delete module;                          // ❻
}


// Is called right after the module is created.  All one time Setup
// should be done now.
void MyEventSegment::Initialize()
{
  module->reset();                       // ❼
  Clear();

}


// Is called after reading data buffer
void MyEventSegment::Clear()
{
  module->clearData();          // Clear data buffer ❽
}

unsigned int MyEventSegment::MaxSize()
{
  return 0;
}


//Is called to readout data on module
DAQWordBufferPtr& MyEventSegment::Read(DAQWordBufferPtr& rBuf)
{
  for(int i=0;i<CAENTIMEOUT;i++)        // ❾
                 // Loop waits for data to become ready
    {
      if(module->dataPresent())
                // If data is ready stop looping
{
  break;
}
    }
  if(module->dataPresent())
               // Tests again that data is ready
    {
      rBuf = m_MyPacket.Begin(rBuf);     // (10)
                // Opens a new Packet

      module->readEvent(rBuf);           // (11)
```

```
                // Reads data into the Packet

      rBuf= m_MyPacket.End(rBuf);       // (12)
                // Closes the open Packet
    }
  return rBuf;                          // (13)
}
```

❶ The lines beginning with the include of `config.h` aneending with the #endif near here are boilerplate that is required for all implementation (.cpp) files for Readout skeletons at version 8.0 and later.

❷ In order to get access to the class definition, we must include the header that we wrote that defines the class

❸ Recall that we will be putting our event segment into a packet. The packet will be managed by a documented packet (`CDocumentedPacket`) object. This packet can document the revision level, or version of the structure of its body. The string `pPacketVersion` will document the revision level of the packet body for our packet.

❹ This code in the constructor is called an *initializer*. Initializers are basically constructor calls that are used to construct base classes and data members of an object under construction. The *`Id`* constructor parameter is used as the id of the `m_MyPacket CDocumentedPacket`. The three strings that follow are, respectively, a packet name, a packet description and the packet body revision level. These items are put in the documentation entry for the packet that is created by the packet at the beginning of the run.

❺ The constructor body creates a new `CAENcard` object, assigning a pointer to it to the member variable `module`. The constructor's *`slot`* is used as the geographical address of the module.

❻ The destructor body destroys the `CAENcard` object created by the constructor. If this is not done here, then everytime we destroy an `MyEventSegment` class, we will "lead" some memory.

❼ This code initializes the CAEN V785 module by resetting it to the default data taking settings, and then invoking our `Clear` member function to clear any data that may be latched.

❽ This code clears any data that is latched in the module

❾ The CAEN V785 requires about 6textmus to convert once it is given a gate. It is possible for the software to start executing the `Read` member function before conversion is complete. This loop repeatedly invokes `dataPresent` for the module. This function returns `true` when the module has buffered conversions. At that time, control breaks out of the loop.

(10) This call to the `Begin` member of our documented packet indicates that we are starting to read data into the body of the packet. The code reserves space for the word count, and writes the packet id. The return value is a "pointer" to the body of the packet.

(11) Data are read from the ADC into the packet.

(12) The size of the packet is written to the space reserved for it. The packet is closed, and a "pointer" is returned to the next free word in the buffer.

(13) a "pointer" to the next free spot in the buffer is returned to the caller.

The numbers below refer to the circled numbers in the example above.

2.1.2. Integrating your event segment with Readout

Once you have created one or more event segments, you must register them with the Readout software. Whenever the Readout software must so something to its event segments, it calls the appropriate member function in each event segment that has been registered, in the order in which it has been registered.

Edit `Skeleton.cpp`. Towards the top of that file, after all the other #include statements, add:

```
#include "MyEventSegment.h"
```

This is necessary because we will be creating an object of class `MyEventSegment`.

Next, locate the function `CMyExperiment::SetupReadout` Modify it to create an instance of `MyEventSegment` and to register that instance as in the italicized code below

```
void
CMyExperiment::SetupReadout(CExperiment& rExperiment)
{
   CReadoutMain::SetupReadout(rExperiment);

   rExperiment.AddEventSegment(new MyEventSegment(10, 0xff00));
}
```

This code creates a new event segment for a CAEN V785 in slot 10, which will be read out into a packet with ID 0xff00.

2.1.3. Making your Readout Executable

Now that you have an event segment written, created an instance of it and registered it with the Readout, you must build an executable Readout program.

To build the program, you must edit the Makefile supplied with the skeleton code so that it knows about your additional program files. Locate the line that reads:

```
Objects=Skeleton.o
```

Modify it so that it reads:

```
Objects=Skeleton.o  MyEventSegment.o
```

Save this edit, exit the editor and type:

```
make
```

This will attempt to compile your readout software into an executable program called `Readout`. If the **make** command fails, fix the compilation errors indicated by it and retry until you get an error free compilation.

2.1.4. Testing the Readout Software

The Reaedout program must run on a system that is physically connected to the VME crate containing your digitizers. Note that the ReadoutShell tool can be used to wrap your program with a Graphical user interface (GUI), and run it on the requested remote system. While debugging your software we do not recommend this. It's better just to run the software from the command line, or under control of the gdb symbolic debugger (see **man gdb** for more information about this extremely powerful debugging tool).

In this section we'll show how to use the command line to

- Start up a very simple low level event dumping program.
- Start your Readout program, start a run, stop a run and exit.
- Interpret the event dumps created while the run is active

There are two buffer dumping programs that are now available to support the NSCL Data acquisition system. Unfortunately, both are called bufdump and can only be distinguished by where they are installed.

`/usr/opt/daq/current/bufdump`

> Dumps the front segments of buffers in real-time. This is the program we will use for our initial testing.

`/usr/opt/utilities/current/bin/bufdump`

> Provides formatted dumps of complete buffers. Each buffer must be individually requested from the data taking system, or an offline file. This program has extensive online and offline help and may be very useful and powerful to detect and debug problems with more complex event structures than our example, but is overkill for this application. A web based manual for this program is available at http://docs.nscl.msu.edu/daq/bufdump/. A PDF printable manual is available at http://docs.nscl.msu.edu/daq/bufdump/manual.pdf. The program's help menu also has very complete documentation.

To start the simple buffer dump program, therefore, log on to the system connected to the VME crate in which your CAEN V785 has been installed and type:

```
spdaqxx> /usr/opt/daq/current/bin/bufdump
```

Output similar to the following should be printed:

```
bufdump version 2.0 (C) Copyright 2002 Michigan State University all rights reserved
bufdump comes with ABSOLUTELY NO WARRANTY;
This is free software, and you are welcome to redistribute it
under certain conditions; See http://www.gnu.org/licenses/gpl.txt
for details
bufdump was written by:

Ron Fox
Eric Kasten
Added Sink Id 2
```

This output indicates that bufdump has successfully connected to the online system of the local computer.

To start the readout software, form another terminal session to the computer that is connected to your VME crate, set the default directory to where you built your software and start the Readout program.

```
spdaqxx> cd ~/experiment/readout
spdaqxx> ./Readout
```

When Readout starts, it should output a message like:

```
pReadout version 1.0 (C) Copyright 2002 Michigan State University all rights reserved
pReadout comes with ABSOLUTELY NO WARRANTY;
This is free software, and you are welcome to redistribute it
under certain conditions; See http://www.gnu.org/licenses/gpl.txt
for details
pReadout was written by:

Ron Fox
%
```

The Readout program you have built runs an extended Tcl interpreter. Tcl is a scripting language that is widely used throughout the NSCL as an extension language. It provides a common base language for *programming* applications. Each program in turn extends this language with a set of commands that allow you to control the application itself.

The commands we will be working with are:

begin

Begins a run. From the point of view of your code, your the `Initialize` and `Clear` member functions of your event segment will be called. The Readout framework will respond to each trigger by calling the `Read` member function of your event segment.

end

Ends a run. The readout framework stops responding to triggers.

Tell the readout program to start a run:

```
% begin
```

The >bufdump terminal session should now start spewing stuff at you. If the trigger rate is high, you may need to stop the run to be able to read the buffers. To do this type: **end** to the Readout program.

When the run starts, bufdump dumps the headers of several non-event buffers. As each event buffer is filled with data collected by the `Read` member function in your event segment, bufdump will dump the buffer header and the first event in the buffer. This might look a bit like this:

```
---------------------- Event (first Event) ----------------
Header:
300 1 -27712 0 1859 0 4 0 0 0 5 258 772 258 0 0
Event:
71(10) words of data
47 46 ff00 4200 2000 4000 404f 4010
4055 4001 4066 4011 404c 4002 4059 4012
4059 4003 4058 4013 404d 4004 405d 4014
405e 4005 4048 4015 4058 4006 4076 4016
4065 4007 4bcc 4017 4050 4008 4068 4018
406b 4009 4057 4019 405a 400a 4067 401a
406a 400b 404e 401b 405a 400c 406c 401c
4066 400d 405a 401d 404f 400e 406e 401e
405a 400f 4055 401f 404c 4404 5824
```

Focus on the section below the word `Event:`. This is a hexadecimal representation of the data in the first event of the buffer. The event that we made starts with 47, this is the total number of words (in hex) that exist in the event. The size of the event is filled in by the Readout framework. The next word is the number of words in our packet (in hex). This word is filled in by our documented packet. The third word is the packet id in this case 0xff00 This word is also filled in by our documented packet object. The remainder of the event is the data from the ADC hardware. The next two words are the ADC header; if you break these down into binary representation the first five bits will be the slot where the V785 is located. If the packet ID and slot number are both correct then you are ready to move on to modifying the SpecTcl Skeleton.

Complete documentation of the data read from the ADC hardware is given by the CAEN V785 manual. This manual can be found online at CAEN website: http://www.caen.it/

Exit the Readout program by typing the following to the Readout program:

```
% end
```

2.2. SpecTcl Histogramming Software

The procedure for setting up SpecTcl parallels that of settting up the Readout software:

- Copy the skeleton for SpecTcl into an empty directory
- Modify this skeleton to unpack the raw events SpecTcl gets from the online system or from eventfiles into a set of parameters.
- Modify the `Makefile` to incorporate your changes into a tailored SpecTcl, and compile the tailored SpecTcl
- Test your SpecTcl on event data acquired by your Readout software.

2.2.1. How to copy the SpecTcl skeleton

At the NSCL, current versions of the SpecTcl skeleton are located in the directory `/usr/opt/spectcl/current/Skel`. The following commands will create a new directory and obtain a copy the SpecTcl skeleton, called MySpecTclApp:

```
mkdir -p ~/experiment/spectcl
```

```
cd ~/experiment/spectcl
cp /usr/opt/spectcl/current/Skel/* .
```

2.2.2. How to modify the skeleton to unpack events.

SpecTcl relies on a logical pipeline of *event processors* to unpack data from the raw event into parameters. Each event processor has access to the raw event, and an output array-like object. This provides each event processor with access to the raw event and any data unpacked by event processors that are ahead of it in the pipeline.

Daniel Bazin has written an extension to SpecTcl called TreeParam. This extension superimposes structure on top of the array-like output object which makes access to the event array much more natural. Beginning with SpecTcl version 3.0, the TreeParam extension has become a supported part of SpecTcl.

Dr. Bazin's TreeParam software also includes a very nice front end Graphical User Interface (GUI) for SpecTcl that makes the creation and manipulation of spectra, gates and other SpecTcl object much more user-friendly than raw SpecTcl. This GUI was also incorporated into the base SpecTcl program beginning with version 3.0, it has been extensively modified for version 3.1.

In order to unpack data from our event segment, we need to write an event processor that:

- Declares the appropriate tree parameter members.
- Locates a packet with our packet id (0xff00).
- Unpacks the data in that packet into the appropriate treeparameter members.

Our event processor `MyEventProcessor` is defined in the file `MyEventProcessor.h`, and implemented in `MyEventProcessor.cpp`. The header looks like this:

Example 2-3. `MyEventProcessor.h` - header for the event processor.

```
#ifndef __MYEVENTPROCESSOR_H
#define __MYEVENTPROCESSOR_H

#include <EventProcessor.h>                    // ❶
#include <TranslatorPointer.h>                 // ❷
#include <histotypes.h>                        // ❸


// Forward class definitions:

class CTreeParameterArray;
```

```
class CAnalyzer;                  // ❹
class CBufferDecoder;
class CEvent;



class MyEventProcessor : public CEventProcessor // ❺
{
private:
  CTreeParameterArray&  m_rawData;              // ❻
  int                   m_nId;                  // ❼
  int                   m_nSlot;                // ❽
public:
  MyEventProcessor(int         ourId,
   int         ourSlot,         // ❾
   const char* baseParameterName);
  ~MyEventProcessor();                    // (10)

  virtual Bool_t operator()(const Address_t pEvent,
    CEvent&         rEvent,   // (11)
    CAnalyzer&      rAnalyzer,
    CBufferDecoder& rDecoder);

private:
  void unpackPacket(TranslatorPointer<UShort_t> p);  // (12)

};

#endif
```

The numbers in the discussion below refer to the numbers in the example above.

❶ All event processors must be derived from the base class `CEventProcessor`. To do this requires that we include the header for that class.

❷ The `TranslatorPointer` class creates pointer-like objects that will do any byte order transformations required between the system that created the event files and the system running SpecTcl. This class is a template class, therefore it is necessary to include the header for it as one of the parameters to a member function of ours is a `TranslatorPointer`<UShort_t>.

❸ The `histotypes.h` header defines types that insulate SpecTcl from differences in the data types in each system. We use these extensively in the header and therefore must include the header here.

❹ Where the compiler does not need to know the internal definitions of a class in a header it is a good idea to create *forward references* to those classes. This can be done when all objects of a class are pointers or references. Doing this decreases the likelyhood that we wind up with a system that has circular include references.

❹ As previously mentioned, our event processor class must be derived from a `CEventProcessor`. This class definition does that.

❻ The TreeParameter package provides both individual parameters (`CTreeParameter`), and arrays of parameters, (`CTreeParameterArray`). In this application we will create a `CTreeParameterArray` that will have one element for each channel of the CAEN V785. This data member will be a *reference* to that array. A reference in C++ is a variable that operates like a pointer with the notation of something that is not a pointer. For example, if I have a pointer p to a structure containing an element b, I would access that element using the notation: p->b. If I have a reference r to the same structure, I would access element b using the notation r.b.

❼ The `m_nId` member variable will be used to hold the id of the packet our event processor is supposed to decode.

❽ The `m_nSlot` member variable will be used to hold the geographical address of the CAEN V785 we are unpacking. This member is used as a *sanity check*. Sanity checks double check that the code you write is operating the way you think it should. In this case, if the packet that matches our id does not contain data that is from the ADC with the correct geographical address, we know that something has gone seriously wrong. It is a good practice to employ sanity checks whenever you can think of one.

❾ The constructor is called to initialize the members of an object of class `MyEventProcessor`. It's parameters are as follows:

`ourId`

Provides the packet id that this object will locate and unpack. By parameterizing this, our class can unpack more than one packet id as long as the packet body is data from a single CAEN V785 ADC

`ourSlot`

Provides the slot number of the ADC we expect to see in the packet of type *`ourId`*. This enables us to perform the sanity check we described earlier.

`baseParameterName`

`CTreeParameter` objects have a parameter name, this parameter name becomes the SpecTcl parameter name. In the case of `CTreeParameterArray` objects, the parameter name becomes a base name and actual parameters have a period and an element number appended to this basename. For example, a ten parameter array with the base name `george` might create SpecTcl parameters `george.00, george.01 ... george.09`. The *`baseParameterName`* will be used to set the `CTreeParameterArray` base name.

(10) A destructor is necessary when classes use dynamcially allocated data, in order to prevent memory leaks when an object is destroyed.

(11) `operator()` allows objects of a class to be treated as functions that can be called. Objects of this sort behave as functions that can retain state between calls. The function call operator of a `CEventProcessor` is a virtual function which means that the appropriate function for pointer to an object derived from `CEventProcessor` will be selected at run time from the actual class of the object the pointer points to (in this case a `MyEventProcessor`. The function call operator for

classes derived from `CEventProcessor` is expected to process the raw event, and already unpacked data producing new unpacked data. The event processor should return a `kfTRUE` if it succeeded well enough that the event pipeline can continue to run or `kfFALSE` if SpecTcl should abort the event pipeline and not histogram the event.

(12) When we implement the `operator()` function, we will have it hunt through the body of the raw event for a packet that matches the id stored in `m_nId` member variable. When we find that packet, we'll want to unpack it into our `m_rawData` tree parameter array. To make the code simpler to understand, we plan to break off the actual unpacking of the packet into a *utility function* called `unpackPacket`.

The next step is to actually implement the class. The implementation of this class is shown in the next listing.

Example 2-4. Implementation of the `MyEventProcessor` class

```
#include <config.h>
#include "MyEventProcessor.h"            // ❶
#include <TreeParameter.h>
#include <Analyzer.h>                    // ❷
#include <BufferDecoder.h>
#include <Event.h>

#include <iostream>

#include <TCLAnalyzer.h>                  // ❸

#ifdef HAVE_STD_NAMESPACE
using namespace std;
#endif
static const ULong_t CAEN_DATUM_TYPE(0x07000000);
static const ULong_t CAEN_HEADER(0x02000000);
static const ULong_t CAEN_DATA(0);
static const ULong_t CAEN_TRAILER(0x04000000);
static const ULong_t CAEN_INVALID(0x06000000);

static const ULong_t CAEN_GEOMASK(0xf8000000);
static const ULong_t CAEN_GEOSHIFT(27);
static const ULong_t CAEN_COUNTMASK(0x3f00);   // ❹
static const ULong_t CAEN_COUNTSHIFT(8);

static const ULong_t CAEN_CHANNELMASK(0x3f0000);
static const ULong_t CAEN_CHANNELSHIFT(16);
static const ULong_t CAEN_DATAMASK(0xfff);


static inline ULong_t getLong(TranslatorPointer<UShort_t>& p)  // ❺
{
  ULong_t high   = p[0];
  ULong_t low    = p[1];
```

```
  return (high << 16) | low;
}

MyEventProcessor::MyEventProcessor(int ourId,
   int ourSlot,
   const char* baseParameterName) :
  m_rawData(*(new CTreeParameterArray(string(baseParameterName),
      4096, 0.0, 4095.0, string("channels"),
      32, 0))),    // ❻
  m_nId(ourId),
  m_nSlot(ourSlot)
{
}



MyEventProcessor::~MyEventProcessor()
{
  delete &m_rawData; // ❼
}

Bool_t
MyEventProcessor::operator()(const Address_t pEvent,
     CEvent&         rEvent,
     CAnalyzer&      rAnalyzer,
     CBufferDecoder& rDecoder)
{
  TranslatorPointer<UShort_t> p(*(rDecoder.getBufferTranslator()),
pEvent);
  UShort_t                    nWords = *p++;           // ❽
  CTclAnalyzer&               rAna((CTclAnalyzer&)rAnalyzer);
  rAna.SetEventSize(nWords*sizeof(UShort_t));
  nWords--; // Remaining words after word count.

  while (nWords) {
    UShort_t nPacketWords = *p;                       // ❾
    UShort_t nId          = p[1];
    if (nId == m_nId) {                               // (10)
      try {                                           // (11)
unpackPacket(p);                                 // (12)
      }
      catch(string msg) {
cerr << "Error Unpacking packet " << hex << nId << dec
     << " " << msg << endl;
return kfFALSE;                                  // (13)
      }
      catch(...) {
cerr << "Some sort of exception caught unpacking packet " << hex << nId
     << dec << endl;
return kfFALSE;
      }
    }
    p      += nPacketWords;                           // (14)
```

```
    nWords -= nPacketWords;
  }
  return kfTRUE;                        // (15)
}

void
MyEventProcessor::unpackPacket(TranslatorPointer<UShort_t> pEvent)
{
  pEvent += 2;                         // (16)

  ULong_t header      = getLong(pEvent);
  if ((header & CAEN_DATUM_TYPE) != CAEN_HEADER) {   // (17)
    throw string("Did not find V785 header when expected");
  }
  if (((header & CAEN_GEOMASK) >> CAEN_GEOSHIFT) != m_nSlot) {
    throw string("Mismatch on slot");
  }

  Long_t nChannelCount = (header & CAEN_COUNTMASK) >> CAEN_COUNTSHIFT;

  pEvent += 2;                                          // (18)
  for (ULong_t i =0; i < nChannelCount; i++) {          // (19)
    ULong_t channelData = getLong(pEvent);
    int     channel     = (channelData & CAEN_CHANNELMASK) >> CAEN_CHANNELSHIFT;
    int     data        = (channelData & CAEN_DATAMASK);
    m_rawData[channel] = data;

    pEvent += 2;
  }

  Long_t trailer = getLong(pEvent);
  trailer       &= CAEN_DATUM_TYPE;
  if ((trailer != CAEN_TRAILER) ) { // (20)
    throw string("Did not find V785 trailer when expected");
  }

}
```

The numbers in the annotations below refer to the numbered elements of the listing above.

❶ Since this compilation unit implements the `MyEventProcessor` class, it is necessary for the file to have access to the header for that class.

❷ The header for `MyEventProcessor` declared several classes to be forward defined. Since these classes will be manipulated in the implementation module, the headers for these classes must now be #include-ed.

❸ The `CAnalyzer` reference that will be passed in to the `operator()` member function is really a reference to a `CTclAnalyzer`. Since we will be calling member functions of `CTclAnalyzer` directly, we must include the header of that class as well

❹ This section of code defines sevaral constants that help us unpack data longwords from the CAEN V785 ADC. See the hardware manual for that module for a detailed description of the format of the buffer produced by this module. Whenever possible avoid *magic numbers* in code in favor of symbolic constants as shown in the listing. Symbolic constants make the code easier to understand for the next person and for you if you have to look at it a long time later. The constants are:

`CAEN_DATUM_TYPE`

This is a mask of of the bits that make up the field in the V785 data that describes what each longword is.

`CAEN_HEADER`

If the bits selected by the `CAEN_DATUM_TYPE` are equal to `CAEN_HEADER` the longword is an ADC header word. Thus you can check for *header-ness* with code like:

```
if ((data & CAEN_DATUM_TYPE) == CAEN_HEADER) { ...
```

`CAEN_DATA`

`CAEN_DATA` selects ADC data longwords in a mannger analagous to the way that `CAEN_HEADER` selects header words. If the bits of a longwords of V785 data are anded with `CAEN_DATUM_TYPE`, and the result is `CAEN_DATA` the longword contains conversion data.

`CAEN_TRAILER`

If a longword of CAEN V785 data, anded with the `CAEN_DATUM_TYPE` mask is `CAEN_TRAILER`, the longword is an ADC trailer and indicates the end of an event from the module.

`CAEN_GEOMASK` and `CAEN_GEOSHIFT`

These two values can be used to extract the *Geographical Address* field from data returned by the CAEN V785 ADC. The Geographical Address is what we have been loosely calling the slot of the ADC. The following code snippet is a recipe for getting the geographical address from any of the longwords that the module returns:

```
unsigned long slotNumber = (data & CAEN_GEOMASK) >> CAEN_GEOSHIFT;
```

`CAEN_COUNTMASK` and `CAEN_COUNTSHIFT`

These two values can be used to return the number of channels present in an event from the CAEN V785 adc. The number of channels present in the event is available in the header word for the ADC data. Given a header word, this information can be extracted as follows:

```
unsigned long channelCount = (header & CAEN_COUNTMASK) >> CAEN_COUNTSHIFT;
```

`CAEN_CHANNELMASK` and `CAEN_CHANNELSHIFT`

These two values are used to extract the channel number of a conversion from a conversion word (type `CAEN_DATA`). The code below shows how to do this:

```
unsigned long channelNumber = (datum & CAEN_CHANNELMASK) >> CAEN_CHANNELSHIFT;
```

`CAEN_DATAMASK`

`CAEN_DATAMASK` can be used to extract the conversion value from an adc conversion longword (longword with type `CAEN_DATA`). For example:

```
unsigned long conversion = datum & CAEN_DATAMASK;
```

❺ The *inline* function `getLong` returns a longword of data pointed to by a `TranslatorPointer`<UShort_t>. The longword is assumed to be big endian by word, that is the first word contains the high order bytes. The bytes within each word are appropriately managed by the `TranslatorPointer` object. Inline functions are preferred to #define macros in C++. The usually have the same execution efficiency without some of the strange argument substitution problems that a #define macro may have.

❻ The main task of the constructor is to initialize data elements. In C++ where possible, this should be done with initializers, rather than with code in the body of the constructor. Note, however, that regardless of the order of initializers in your constructor, they are executed in the order in which the members *are declared in the class definition*. The only initializer worth explaining in this code is the one for the `CTreeParameterArray&` member data `m_rawData`. This constructor dynamically allocates a new object and points the reference at it. The parameters of the constructor are in turn the base name of the array, The preferred number of bins for spectra created on this parameter, the low and high limits of the range of values this parameter can have, the units of measure of the parameter, the number of elements in the array and the index of the base element of the array (it is possible to give a `CTreeParameterArray` any integer base index desired.

❽ This section of code should appear in essentially all event processor `operator()` functions. `p` is an object that acts very much like a UShort_t*. However it transparently does any byte order swapping required between the binary representations of the system that created the buffer and the system that is runing SpecTcl. This "pointer" is then used to extract the size of the body of the event from the buffer. One of the responsibilities of at least one event processor is to report the size of the event in bytes to the SpecTcl framework. The `CAnalyzer` object reference `rAnalyzer` is actually a reference to an object of type `CTclAnalyzer`. This object has a member function called `SetEventSize` which allows you to inform the framework of the event size. Since any event processor can and in many cases should be written to run with little or no knowledge of other event processors in the pipeline, by convention we have every event processor informing the framework of theevent size in this way.

❾ This code extracts the packet size and id of each packet encountered in the while loop. The size is used to skip to the next packet in the event.

(10) This if checks for packets that match the id we've been told to unpack (`m_nId`).

(11) A try .. catch block in C++ executes code in the body of the try block. If the code raises an exception, a matching exception handling catch block is searched for at this an higher call levels. If

found that catch block's code is executed. If not found, the program aborts. This block allows us to catch errors from the `unpackEvent` function and turn them into pipeline aborts.

(12) If the event we are unpacking contains the packet our processor has been told to match, `unpackPacket` is called to unpack the body of our packet into `m_rawData`.

(13) If `unpackPacket` fails a sanity test it will throw a string exception. This catch block will then execute, printing out an appropriate error message to `stderr` and returning `kfFALSE` which will cause the remainder of the event processing pipeline to be aborted, and the event to be thrown out by the histogrammer. We have also supplied a *catch all* catch block (catch (...)). If an unexpected exception is thrown from member functions called by `unpackPacket` (for example `TranslatorPointer` members), this will report them as well.

(14) The packet size is used to point `p` to the next packet in the event, until we run out of packets.

(15) Returns `kfTRUE` indicating that SpecTcl can continue parameter generation with the next element of the event processing pipeline or commit the event to the histogramming subsystem if this was the last event processor.

(16) This code in `MyEventProcessor::unpackPacket` adjusts the `TranslatorPointer pEvent` to point to the body of the packet. This should be the first word of data read from the ADC itself. For the CAEN V785, this should be a longword of header information.

(17) Two sanity checks are performed on the header. First the type of the longword is analyzed and an exception is thrown if the longword is not a header longword. Second, the geographical address of the V785 is extracted and a string exception is thrown if the header is not the header for the geographical address of the module we've been told to expect (`m_nSlot`). If these sanity checks pass, the number of channels of event data read by the module for the event are extracted from the header and stored in `nChannelCount`.

(18) Once it's clear that the header is really a header for the correct module, we adjust `pEvent` to point to the next longwors which should be the first of `nChannelCount` longwords containing conversion information.

(19) This loop decodes the `nChannelCount` longwords of data. The channel number and conversion values are extracted, and stored in the appropriate element of the `m_rawData CTreeParameterArray`. A useful exercise for the reader would be to add an appropriate sanity check for this section of code. One useful check would be to ensure that the data words really are data words, that is that their type is `CAEN_DATA`.

(20) Ensures that the longword following the last channel data longword is a V785 trailer longword.

Once we have written our event processor, we need to create an instance of it and add it to the event processing pipeline. This is done by modifying the `MySpecTclApp.cpp` file. Edit this file and locate the section of code where header files are #include-ded. Add the italicized line as shown.

```
#include <config.h>
#include "MySpecTclApp.h"
#include "EventProcessor.h"
#include "TCLAnalyzer.h"
#include <Event.h>
#include <TreeParameter.h>
```

```
#include "MyEventProcessor.h"
```

This makes the definition of the event processor we created known to this module.

Locate the code that creates the analysis pipeline. In the unmodified `MySpecTclApp.cpp` file this will look like:

```
void
CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{

#ifdef WITHF77UNPACKER
  RegisterEventProcessor(legacyunpacker);
#endif

    RegisterEventProcessor(Stage1, "Raw");
    RegisterEventProcessor(Stage2, "Computed");
}
```

Rewrite this section of code so that it looks like this:

```
void
CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
  RegisterEventProcessor(*(new MyEventProcessor(0xff00, 10,
"mydetectors.caenv785")));
}
```

This creates a new event processor that will look for Id 0xff00, expect it to contain data from geographical address 10, and unpack this data into the tree parameter array with a base name of mydetectors.caenv785.

2.2.3. Building the tailored SpecTcl

To build our the SpecTcl we have tailored for our events, we need to modify the `Makefile` to make it aware of our event processor, so that it will compile and link it into our SpecTcl.

Edit `Makefile`. Locate the line that reads:

```
 OBJECTS=MySpecTclApp.o
```

Modify it to read as follows:

```
OBJECTS=MySpecTclApp.o MyEventProcessor.o
```

Now type:

```
make
```

to build the program. This should result in an executable file called `SpecTcl`.

2.2.4. Setting up SpecTcl Spectra

Run the version of SpecTcl you created. Four windows should pop up:

Xamine

Is a window that allows you to view and interact with the spectra that SpecTcl is histogramming.

TkCon

This is a command console that is modified from the work of Jeffrey Hobbs currently at ActiveState. You can type arbitrary Tcl/Tk and SpecTcl commands at this console. This software was released into the open source world under the BSD license.

SpecTcl

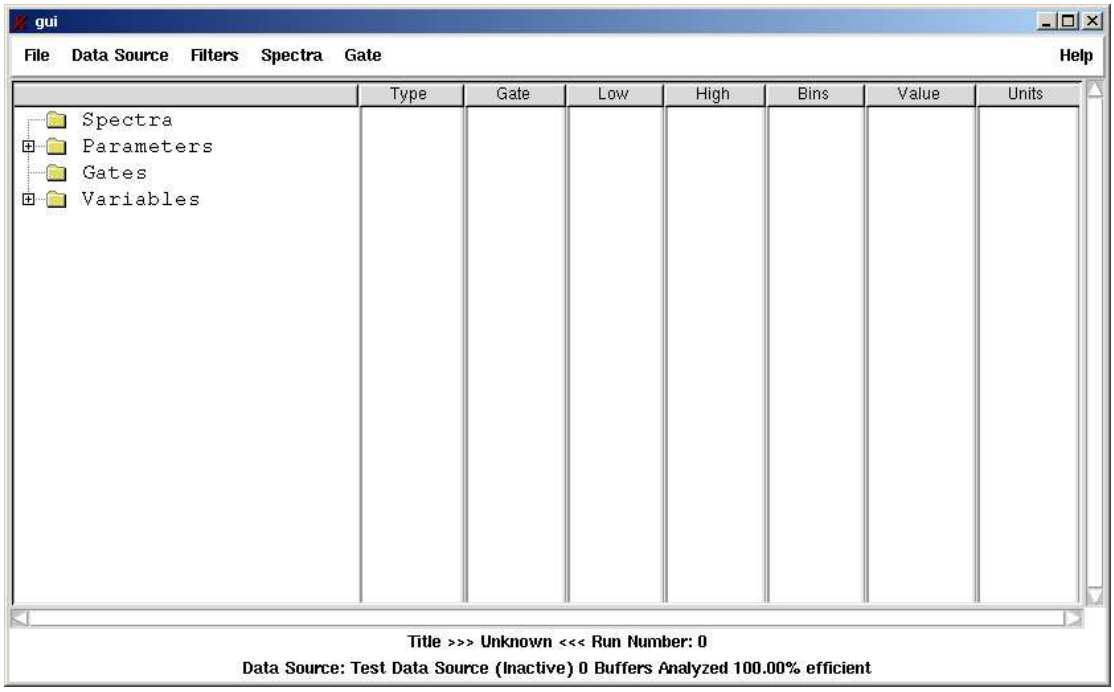
Is a button bar that is built on top of the Tk top level widget. Untailored, it has two buttons, *Clear Spectra* clears the counts in all spectra. *Exit* exits SpecTcl.

gui

Is a tree browser based interface that allows you to manipulate SpecTcl's primary objects, Spectra, Parameters, Gates, and Variables. The idea for this sort of interface is Daniel Bazin's, this interface, however does not bear any common code nor resemblence to his original work.

We will interact with the gui window to create an initial set of 1-d spectra, one for each channel of the adc. First lets look at the figure below.

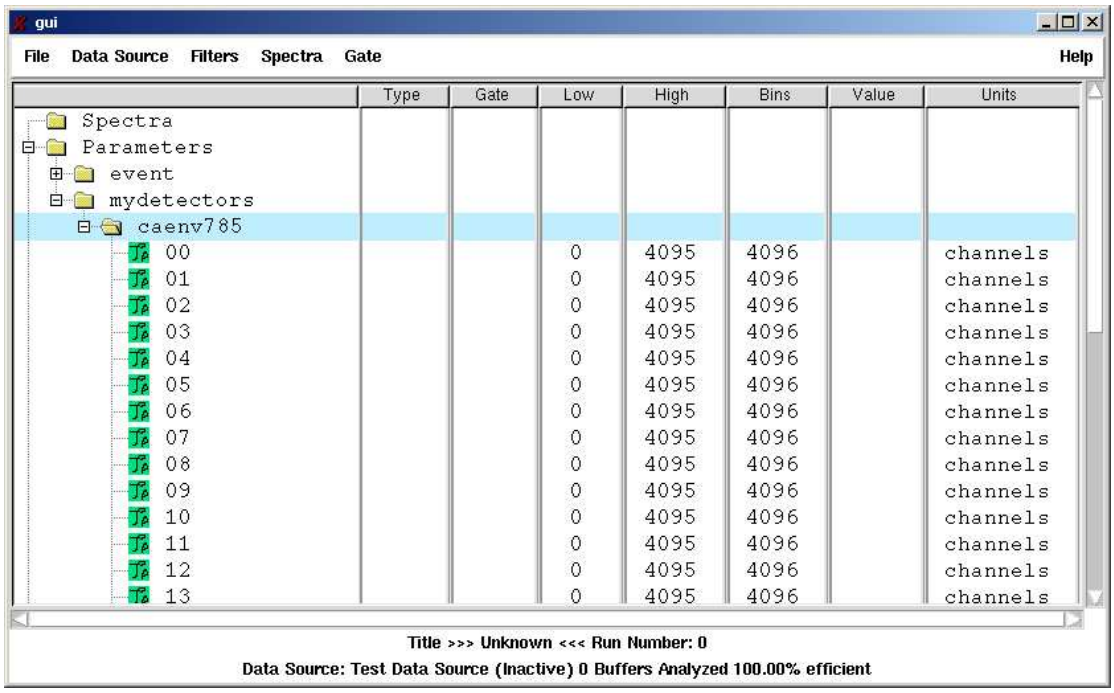
Figure 2-1. The GUI window as it first appears.



This shows a screenshot of the GUI just after it starts. Each top level folder contains or will contains the set of objects named by the folder. The Spectra folder will contain spectra. Folders that have a + to the left of them already contain objects. Since we did not delete the example tree parameters and variables in `MySpecTclApp.cpp` TreeVariable objects already exist. Parameters have been defined, both by our event processor and by the samples in the oritinal `MySpecTclApp.cpp`

Click on the + to the left of the Parameters folder. You should see two subfolders. named `event` and `mydetectors`. The first of these contains parameters that were created by the examples in `MySpecTclApp.cpp`. The second contains parameters we created in our event processor. Recall that our basename was mydetectors.caenv785, the gui creates a new folder level for each period in a name. Double clicking on mydetectors opens it revealing the caenv785 subfolder we expect. This subfolder contains all the elements of the `CTreeParameterArray` that was created when we constructed our event processor. Double clicking on the caenv785 subfolder gives the following:

Figure 2-2. The Gui with folders open to show parameter array elements

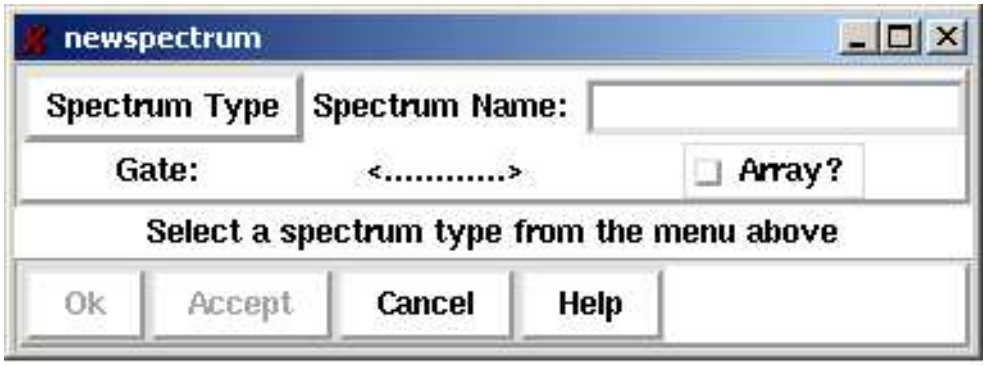


You can use the scrollbar at the right of the window to scroll up and down the through the list of parameters. You can also resize the window to make more parameters show simultaneously.

We will now use the Spectra folder *context menu* to create an array of spectra, one for each of the parameters in the array mydetectors.caenv785. Every folder has a context menu, as do most objects. A context menu is a menu that pops up under the mouse when you hold downt the right mouse button. Context menu entries are selected by moving the mouse pointer over one of the menu entries and releasing the right mouse button.

Select the New... context menu entry from the Spectra context menu. This brings up a new window shown in the figure below:

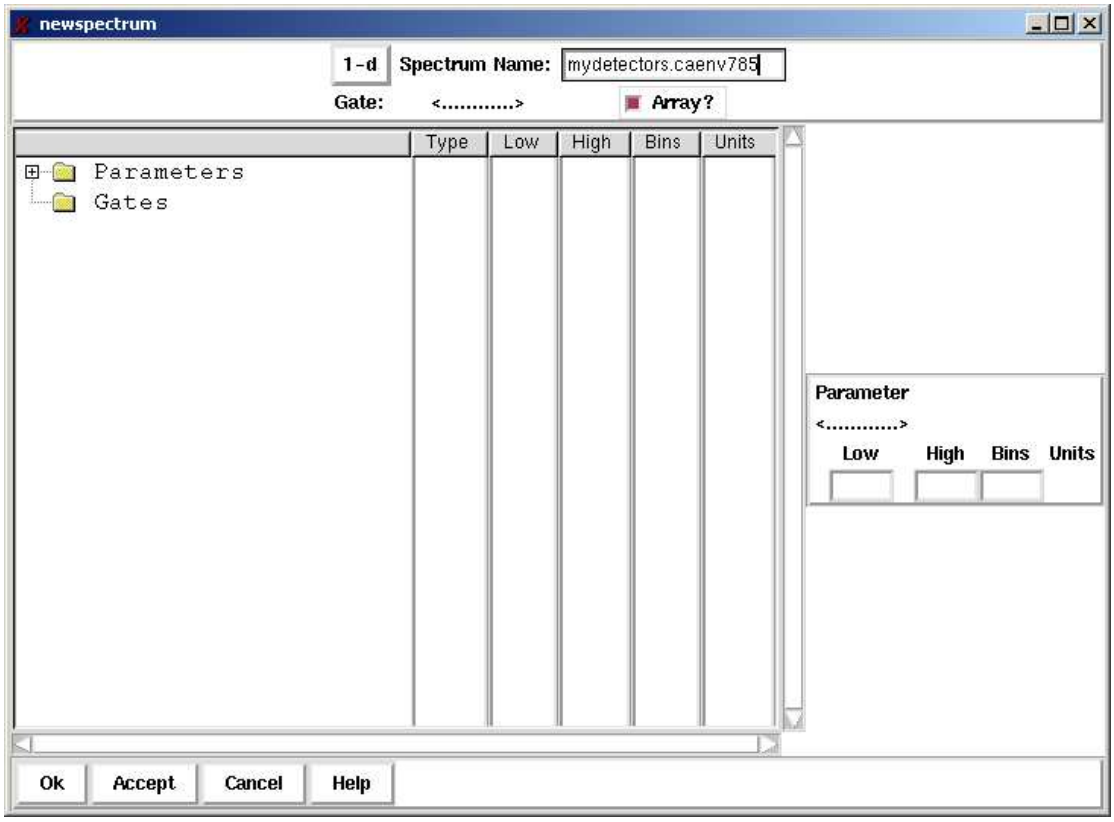
Figure 2-3. The Spectrum creation dialog box



This dialog has several controls. The button labelled Spectrum Type accesses a pull down menu of spectrum type choices. The entry next to the label Spectrum Name: provides a space to type in the name of the spectrum. The Array? checkbutton is an idea from the original Tree Parameter Gui by Dr. Bazin. This button allows you to simultaneously create 1-d spectra for each element of a parameter array, by providing a definition for a spectrum for a single element. We'll use this.

Type mydetectors.caenv785 in the Spectrum Name: entry. Check the Array? checkbutton. Pull down the Spectrum Type menu and select 1-d. The spectrum creation dialog modifies itself to be a 1-d spectrum editor:

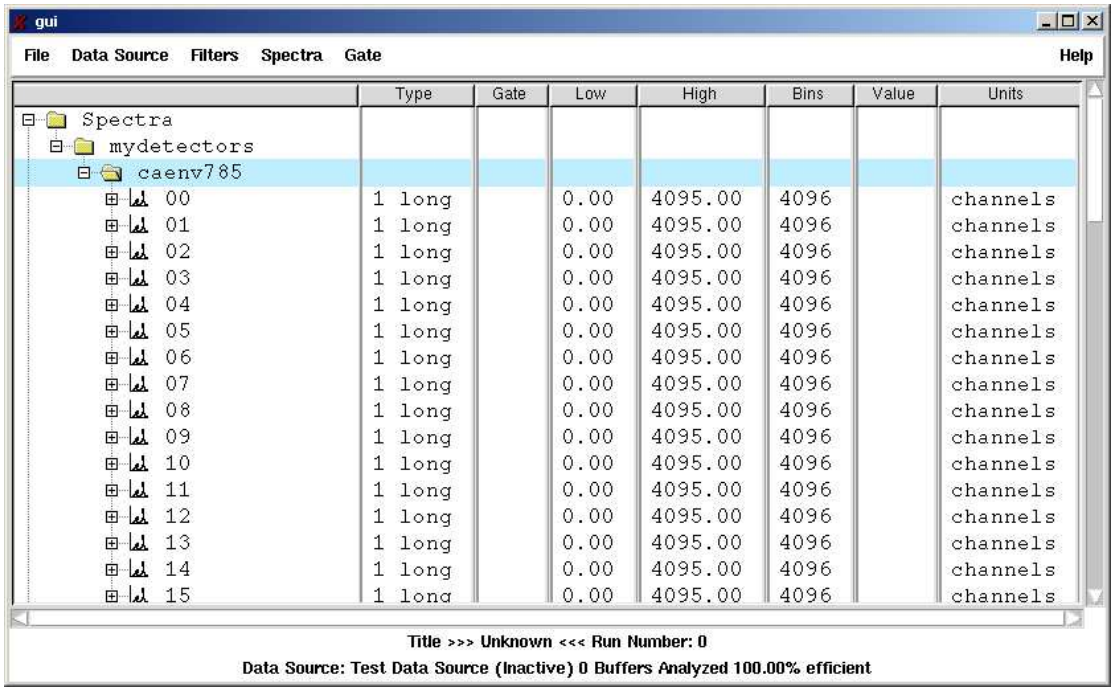
Figure 2-4. 1-d Spectrum editor.



The left half of this editor is just the SpecTcl object browser restricted to display only the set of objects that are relevant to constructing spectra, the parameters and the gates. A parameter is required, a gate is optional. Open the Parameters folder to display the mydetectors.caenv785 parameter array. The double click on any parameter from that array, say mydetectors.canv785.00. This parameter definition is loaded into the right side of the window. You can type modifications to the range and binning of the histogram. For now, we'll accept the defaults. Create the spectrum by clicking the Ok button at the bottom of the 1-d spectrum editor.

Note that in the main gui window, there is now a + to the left of the Spectra top level folder indicating that spectra have been defined. Open this folder to reveal the spectra we just made:

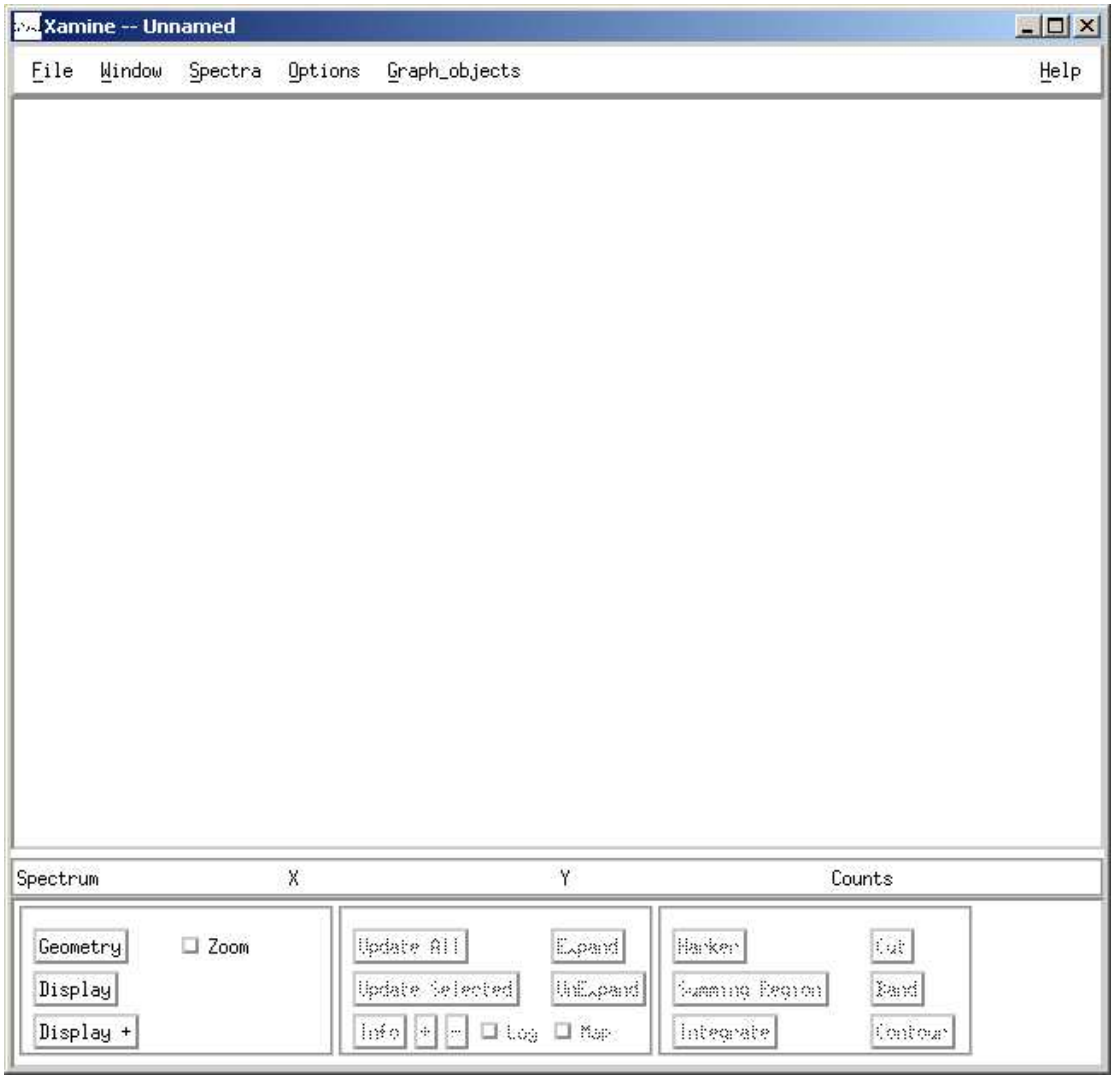
Figure 2-5. Created Spectra.



While this is not hard to do, it is annoying to have to do this each time we start SpecTcl. We will therefore save the current set of definitions in a definition file. On the GUI click the File⟶Save Menu entry. A file choice dialog will pop up. Type myspectra in its File name: entry and click Save.... From now on, whenever you start SpecTcl, you can load these definitions, by clicking the File⟶Restore... menu entry and selecting the `myspectra.tcl` file.

Next we'll want to setup Xamine so that we can actually see the spectra we create. The Xamine window looks like this:

Figure 2-6. Initial Xamine Window



The top part of the Xamine window has a menubar. The large central section has can be subdivided into panes. One (or more in the case of compatible 1-d) spectrum can be loaded into each pane. Below the spectrum display area is a status bar. The status bar will display information about the mouse pointer when it is located in a pane that has been loaded with a spectrum. Below the status bar, a set of commonly used functions are organized into several groups of buttons.

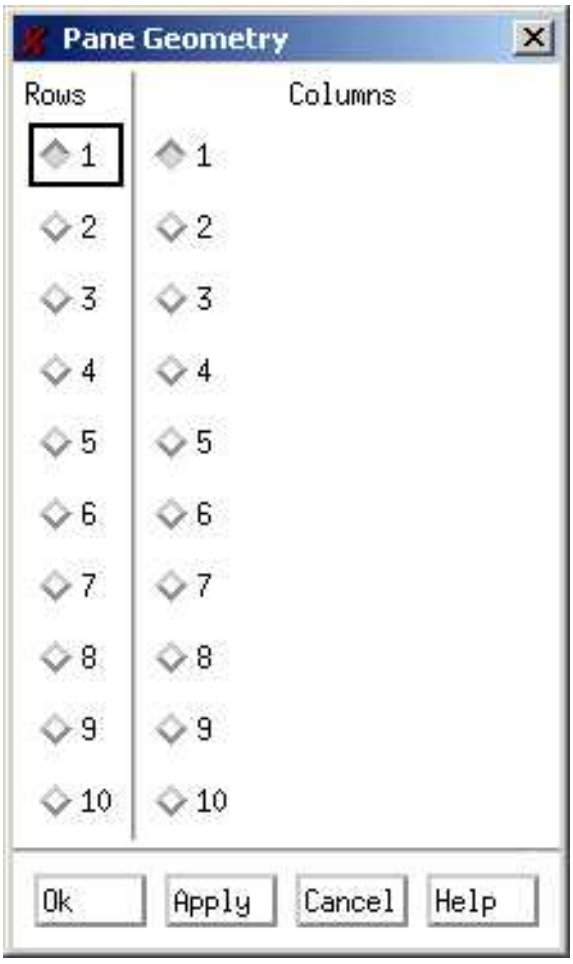
We will now:

- Subdivide the spectrum display area into an array of panes, four rows down, by eight columns across.
- Load the 32 1-d spectra we have just created into the panes.

- Save the resulting configuration file as a *window definition file* so that it can be loaded again next time SpecTcl is run.

Locate the button in the left most box of buttons at the bottom of the screen labeled Geometry and click on it. This brings up the following dialog box:

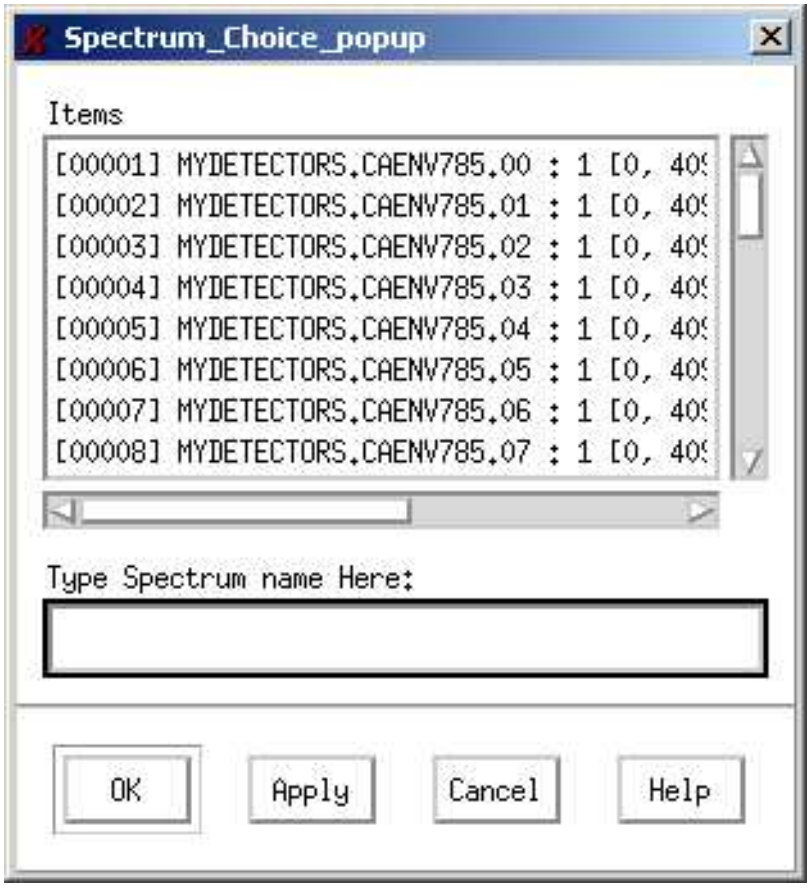
Figure 2-7. The Pane Geometry dialog



Click the 4 button in the left column labeled Rows. Click the 8 button in the right column labeled Columns. Click the Ok button at the bottom of he dialog. The dialog will disappear and the central part of the Xamine window will be subdivided as you requested. Notice how the upper left pane of the set of panes you just created appears pushed in? This pane is said to be *selected*. Many of the operations in the Xamine window operate on the selected pane. A single mouse click in a pane selects it. A double mouse click zooms the pane to fill the entire window. A double mouse click on the expanded pane unzooms the pane.

Lets load the spectrum mydetectors.caenv785.00 into the upper left pane. Click on the upper left pane to select it. Click the Display button. This brings up the spectrum choice dialog shown below:

Figure 2-8. Spectrum Choice dialog



The Xamine title for a spectrum includes additional information about the spectrum. The leftmost part of the title, however is the spectrum name. Double click the top most entry `[1] MYDETECTORS.CAENV785.00 : 1 [0, 4095 : 4096] {MYDETECTORS.CAENV785.00}`. The spectrum chooser dialog disappears and the spectrum you selected is loaded into the upper left pane, which remains selected. Double click it to zoom, double click again to unzoom.

You could repeat this action 31 more times to load each pane with the desired spectrum. That would be painful. Loading a set of panes is a common enough activity that Xamine includes built in support for it. Click the second pane from the top left to select it. Click the Display + button. This brings up another Spectrum Chooser dialog. Single click on the second spectrum `MYDETECTORS.CAENV785.01 : 1 . . . .` Hit the enter key. Note that:

- The dialog remains visible.

- The selected pane is loaded with the spectrum you selected.
- The selection advances to the next pane.

You can finish loading the panes by hitting the down arrow key to select the next spectrum and hitting enter. Repeat as needed to fill all the panes. When all panes are full, click the Cancel button to dismiss the dialog.

Now click on the Window⟶Write Configuration... menu selection. This will bring up a file selection dialog box. In the entry box labelled Filename:, append the text: `MyWindows` and click Ok. This creates a new file `MyWindows.win` that can be used to reload the window configuration you have just created. Window definition files, as these files are called, can be loaded using the Window⟶Read Configuration... menu selection.

Exit SpecTcl this is done either by clicking the Exit command on the button strip labeled SpecTcl, or by typing **`exit`** in the console window tkcon. Note that the File⟶Exit menu entry on the SpecTcl gui only exits the gui, leaving SpecTcl up and running.

Chapter 3. Testing and Running the Software.

Now let's put all the pieces together and test the system end-to-end. We will:

- Start SpecTcl
- Reload our spectrum and window definition files into SpecTcl
- Connect SpecTcl to the online system so that it will analyze data taken from our Readout program.
- Start the Readout Program
- Start a run in Readout
- Check that we see counts in our spectra in SpecTcl

First lets start SpecTcl and recover the definitions we made earlier. With your working directory set `~/experiment/spectcl`, start SpecTcl by typing: **./SpecTcl**

Once all of the SpecTcl windows pop up, reload the spectrum configuration using: File⟶Restore... on the GUI window. Next load the window definition file for Xamine by clicking: Window⟶Read Configuration.... You should now have an array of spectra in the Xamine window, four rows, by eight columns, as before.

Now lets hook SpecTcl to the online system. SpecTcl has a concept of data sources. A data source is a source of event data that SpecTcl can histogram. The Data Source menu on the SpecTcl Gui provides a selection of data source types. Click the Data Source⟶Online (spectrodaq)... menu item on the Gui window. This will bring up the dialog shown in the following figure.

Figure 3-1. Online source host selection dialog

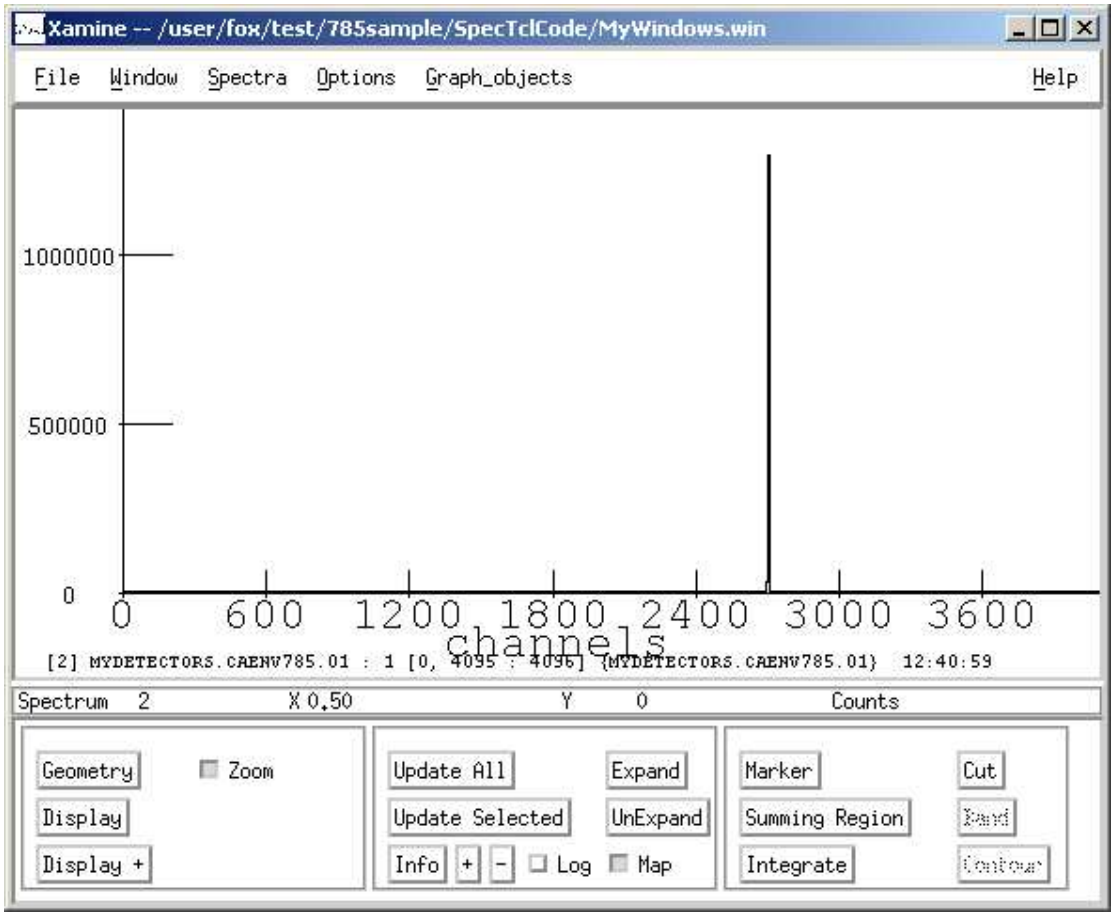
The NSCL Data acquisition system is a distributed system. It is therefore possible to histogram data on a system other than the one the Readout runs on. In most real experiments at the NSCL, we do exactly that. In this case, we will run SpecTcl and the readout program on the same system. Therefore the default host `localhost` is correct. Simply click the Ok button to connect to the online system.

Now we must start Readout and begin a run so there is data for SpecTcl to analyze. With our current working directory set to `cd ~/experiment/readout`, start the readout program: **./Readout**. When you get the prompt, type: **begin**.

The status line at the bottom of the SpecTcl GUI should reflect the number of buffers SpecTcl has analyzed and the analysis efficiency. SpecTcl is not allowed to dictate the data rate. Therefore it only analyzes the fraction of the data it is able to keep up with. The efficiency reflects that percentage. When analyzing data from offline sources (e.g. event files), all data are analyzed, so the efficiency is 100%.

In the Xamine window, click the Update All button in the second box of buttons. The histogram for the channel into which you have put the pulser should show a sharp peak. If you double click this spectrum to zoom in, you should see something like this:

Figure 3-2. Sample Pulser Peak



By adjusting the pulse height you should be able to move this peak across the spectrum. By switching input channels you should be able to produce this peak in any of the histograms.

Chapter 4. More information

If you are preparing a system that will be used many times, you may want to automate the startup as well as run the Readout software under the control of a Readout Gui called ReadoutShell. Normally this automation is done by:

- Writing scripts to start each of the components of the system.
- Attaching these scripts to icons on a KDE desktop.

It is a good idea to get all of your software debugged first, as debugging programs run under desktop icons can be challenging.

4.1. Scripting and desktop icons

The trickiest issue for creating scripts that will become desktop icons is that the environment in which you are running your scripts and programs is not well known. You should ensure that each script sources appropriate login scripts and sets the working directory you want your application to run in. We will generate four scripts:

1. A script to run SpecTcl
2. A script to initialize SpecTcl, loading in our histogram definitions and connecting it to the appropriate online system.
3. A script to run the ReadoutShell Readout GUI.
4. A script that will be used by the ReadoutShell to start the readout program.

Once these scripts are written we will create desktop icons for the first two scripts so that you can start SpecTcl and Readout by clicking on the desktop.

4.1.1. Scripts and a desktop shortcut for SpecTcl

The following script will be used to start SpecTcl:

Example 4-1. SpecTcl startup script.

```
#!/bin/bash

. /etc/profile              # ❶
. ~/.bashrc

cd ~/experiment/spectcl     # ❷

./SpecTcl <setup.tcl        # ❸
```

Refer to the numbers in the listing above when reading the annotations below:

❶ Since icons may or may not source the various bash startup scripts we explicitly source the system wide and our login specific startup scripts.

❷ We set the working directly explicitly to the directory in which we have installed our SpecTcl. This ensures that when SpecTcl is started, it will find its initialization scripts.

❸ This starts SpecTcl, setting its standard input to the file `setup.tcl` located in `~/experiment/spectcl`, the working directory. This file will initialize SpecTcl, and will be written next.

In our startup script for SpecTcl we pointed the SpecTcl stdin at the file `setup.tcl`. This file will setup the initial spectrum definitions and attach SpecTcl to the online system.

The `spectcl.tcl` startup script is as shown below:

Example 4-2. The `spectcl.tcl` SpecTcl startup script

```
source myspectra.tcl;      # ❶
sbind -all;                # ❷

.gui.b update;             # ❸


if {[array names env DAQHOST] ne ""} {
    set daqsource $env(DAQHOST)
} else {;                   # ❹
    set daqsource "localhost"
}

set url "tcp://$daqsource:2602"; # ❺

attach -pipe /usr/opt/daq/current/bin/spectcldaq $url; # ❻
start;                                                 # ❼
```

The numbers in the explanation below refer to the numbers in the example above.

❶ This source command sources the spectrum definitions we created when we configured SpecTcl. SpecTcl GUI configuration files are just Tcl Scripts. Sourcing these scripts reproduces the definitions saved in them.

❷ The **sbind** command here binds all of the spectra into the Xamine visualization program. SpecTcl spectra need not be visible to Xamine, this command ensures they will be.

❸ Unless told to do so, the GUI's object browser will not reflect definitions that have been made outside the gui. This command tells the gui's object browser (widget .gui.b), to rebuild the object tree.

❹ By convention, most experiments use the environment variable `DAQHOST` to store the name of the host on which Readout runs. This code sets `daqsource` to be the value of the `DAQHOST` environment variable if it is defined, or to `localhost` if it is not. This establishes the system from which data will be analyzed.

❺ Connections are specified to remote data acquisition systems using *URL* notation. This command sets `url` to be the url corresponding to the spectrodaq server buffer request socket for the host `daqsource`.

❻ The SpecTcl **attach** command specifies the data source from which SpecTcl will process buffers. In this case we specify that we will accpet buffers from a pipe to which the program spectcldaq in the data acqusition system will be attached. spectcldaq accepts data from the specified source, sampling event data, and dumps the data to its standard output (the other end of the pipe SpecTcl is taking buffers from.

❼ Once SpecTcl is attached to a data source the **start** command starts processing data from that data source.

4.2. Using the Readout GUI ReadoutShell

ReadoutShell provides a graphical user interface front end to the readout application. This front end supports:

- Starts the readout program on an arbitrary host.
- Starting, stopping, pausing, resuming runs
- Setting run parameters such as the title, run number and scaler readout period.
- Enabling or disabling software to start or stop recording event data to disk for later detailed analysis.
- Performing timed runs, that is runs that have a fixed pre-determined run time.
- User programmable actions to be performed when run state transitions are performed, or the Readout program started or restarted.
- Management of data sets and symbolic links so that event data can be treated as grouped with any arbitrary associated data files, or as just a directory containing the event data.

In this section we will create a script to start up the ReadoutShell in such a way that our readout program, `~/experiment/readout/Readout` will be started by it. We will ensure our readout runs on the host

specified by the DAQHOST environment variable, if it is defined, or localhost if not.

Consider the following script:

Example 4-3. Starting ReadoutShell `~/bin/startreadout`

```
#!/bin/bash

. /etc/profile
. ~/.bashrc


if test "$DAQHOST" == ""
then
  host=‘hostname‘               # ❶
else
  host="$DAQHOST"
fi

/usr/opt/daq/8.1/bin/ReadoutShell -host=$host \
  -path=$HOME/experiment/readout/Readout # ❷
```

❶ This shell script code checks to see if the environment variable DAQHOST is undefined or blank. If so, then the shell script variable host is set to the name of the system the script is running on (Readout will be run locally). If not, host is set to the value of the DAQHOST variable.

❷ These two lines start the ReadoutShell GUI telling it to run the readout program we created on the host specified by the host variable.

The readout application we wrote does not need to know what its working directory is, it also does not rely on any environment variables that got set by either the system wide, or user profile scripts (/etc/profile, and ~/.bashrc). It is possible that you might write a readout program that does rely on knowing the working directory, the value of an environment variable or both. In that case, instead of directly running the readout program, as shown in the example above, specify that the readout program is itself a shell script that will source the appropriate profiles and set the appropriate working directory.

4.3. Creating desktop icons

This section is specific to the KDE desktop manager. If you have chosen to use a different desktop manager for your experiment, you will need to figure out how to establish desktop shortcuts on your own.

Creating a desktop shortcut on KDE

1. Login to a Linux desktop in the DAQ network. This could be an spdaq system or it could be a data-U system. Login using the user name and password for the account under which you will want the shortcut defined.
2. Right click on any empty chunk of desktop. This will bring up a *context menu*.
    a. Select the Create New⟶File⟶Link to Application...
3. A dialog will pop up that allows you to create a new desktop short cut:
    a. In the General tab, enter the name of the short cut (this will label the short cut on the desktop), and click the button that's labeled with a blue gear to select an icon image for the shortcut.
    b. In the Application tab, enter the Click the Browse... button next to the entry labeled Command:, browse the filesystem and select the filename for the script you want attached to the shortcut, for example, `~/bin/startreadout`, the most recent script we created.
4. There should now be a new shortcut on the desktop. Click it once to test that it works correctly.

Chapter 5. Complete program listings.

5.1. Readout Software

This section contains the files that make up the Readout software in their entirety. We include the Makefiles and scripts as well as the C++ source code. This software is also available online at http://docs.nscl.msu.edu/daq/samples/CAENV785/CAENV785.zip

Example 5-1. MyEventSegment.h

```
/*
This is the header file to define the MyEventSegment class, which
is derived from CEventSegment.  This class can be used to read
out any number of CAEN modules covered by the CAENcard class.
Those cards include the V785, V775, and  V792.

Tim Hoagland
11/3/04
s04.thoagland@wittenberg.edu

*/


#ifndef __MYEVENTSEGMENT_H                // co:protect
#define __MTEVENTSEGMENT_H
#ifdef HAVE_STD_NAMESPACE
using namespace std;                      // co:namespace
#endif
#include <spectrodaq.h>
#include <CEventSegment.h>
#include <CDocumentedPacket.h>
#include <CAENcard.h>
#define CAENTIMEOUT 50


// Declares a class derived from CEventSegment
class MyEventSegment : public CEventSegment   // co:eventsegderived
{
private:
  CDocumentedPacket m_MyPacket;              // co:docpacket
  CAENcard* module;                          // co:hardwareobject
public:
  MyEventSegment(short slot,unsigned short Id); // co:mysegconstructor
              // Defines packet info
  ~MyEventSegment();                 // co:mysegdestructor

  virtual void Initialize();                    // co:myseginit
```

```
            // One time Module setup

  virtual void Clear();                          // co:mysegclear
            // Resets data buffer

  virtual unsigned int MaxSize();                // co:mysegobsoletemaxsize

  virtual DAQWordBufferPtr& Read(DAQWordBufferPtr& rBuf);  // co:mysegread
            // Reads data buffer
};
#endif
```

Example 5-2. MyEventSegment.cpp

```
/*
    This software is Copyright by the Board of Trustees of Michigan
    State University (c) Copyright 2005.

    You may use this software under the terms of the GNU public license
    (GPL).  The terms of this license are described at:

     http://www.gnu.org/licenses/gpl.txt

     Author:
             Ron Fox
     NSCL
     Michigan State University
     East Lansing, MI 48824-1321
   */


/*

This is the implementation file for the MyEventSegment
class.  This class defines funtions that can be used to
readout any module covered in the CAENcard class.  These
include the V785, V775, and V792


Tim Hoagland
11/3/04
s04.thoagland@wittenberg.edu


*/



#include <config.h>
#ifdef HAVE_STD_NAMESPACE                    // co:boilerplate
```

```
using namespace std;
#endif

#include "MyEventSegment.h"                    // co:myheader

static char* pPacketVersion = "1.0";           // co:packetversion
// Packet version -should be changed whenever major changes are made
// to the packet structure.


//constructor set Packet details
MyEventSegment::MyEventSegment(short slot, unsigned short Id):
  m_MyPacket(Id,
     string("My Packet"),
     string("Sample documented packet"),
     string(pPacketVersion))       // co:packetinit
{
  module = new CAENcard(slot);            // co:cardcreate

}
// Destructor:
MyEventSegment::~MyEventSegment()
{
  delete module;                           // co:destroy
}


// Is called right after the module is created.  All one time Setup
// should be done now.
void MyEventSegment::Initialize()
{
  module->reset();                       // co:moduleInit
  Clear();

}


// Is called after reading data buffer
void MyEventSegment::Clear()
{
  module->clearData();          // Clear data buffer co:cleardata
}

unsigned int MyEventSegment::MaxSize()
{
  return 0;
}


//Is called to readout data on module
DAQWordBufferPtr& MyEventSegment::Read(DAQWordBufferPtr& rBuf)
{
  for(int i=0;i<CAENTIMEOUT;i++)        // co:waitready
```

```
                // Loop waits for data to become ready
    {
      if(module->dataPresent())
                // If data is ready stop looping
{
  break;
}
    }
  if(module->dataPresent())
               // Tests again that data is ready
    {
      rBuf = m_MyPacket.Begin(rBuf);     // co:openpacket
               // Opens a new Packet

      module->readEvent(rBuf);           // co:readdata
               // Reads data into the Packet

      rBuf= m_MyPacket.End(rBuf);       // co:closepacket
               // Closes the open Packet
    }
  return rBuf;                          // co:returnpointer
}
```

Example 5-3. Skeleton.cpp

```
/*
    This software is Copyright by the Board of Trustees of Michigan
    State University (c) Copyright 2005.

    You may use this software under the terms of the GNU public license
    (GPL).  The terms of this license are described at:

     http://www.gnu.org/licenses/gpl.txt

     Author:
             Ron Fox
     NSCL
     Michigan State University
     East Lansing, MI 48824-1321
*/


/*
This file was modified to readout segments from the MyEventSegment class.
It is set to read only one card although that is easily changed.

Tim Hoagland
11/3/04
s04.thoagland@wittenberg.edu
```

```
Modified to be an 8.1 example
Ron Fox
3/28/2006
fox@nscl.msu.edu
*/


static const char* Copyright = "(C) Copyright Michigan State University 2002, All rights res

//  This file contains a test readout system.
//   It derives from the CReadoutMain class
//   to setup our experiment specific requirements
//   creates an instance of it and lets the base classes
//  do most of the work:

#include <config.h>                      // co:config
#include <CReadoutMain.h>
#include <CExperiment.h>
#include <CInterpreterStartup.h>
#include <CInterpreterCore.h>
#include <CRunVariableCommand.h>
#include <CRunVariable.h>
#include <CStateVariableCommand.h>
#include <CStateVariable.h>
#include <TCLInterpreter.h>
#include <CDAQTCLProcessor.h>
#include <CVMEScalerLRS1151.h>
#include <CTraditionalEventSegment.h>
#include <CTraditionalScalerReadout.h>
#include <CEventSegment.h>
#include "MyEventSegment.h" // co:myeventseginclude


#ifdef HAVE_STD_NAMESPACE
using namespace std; // co:stdnamespace
#endif


           //  Added MyEventSegement Header File



/*!
    Sample implementation of an experiment specific1
   tailoring of the production readout software.

*/
class CMyExperiment : public CReadoutMain
{
public:
   // Constructors and other canonical operations:

   CMyExperiment() {
```

```
   }
   virtual ~CMyExperiment() {
   }
   // Copy construction, assignment and comparison
   // make no sense and are therefore disallowed:
private:
   CMyExperiment(const CMyExperiment& rhs);
   CMyExperiment& operator=(const CMyExperiment& rhs);
   int            operator==(const CMyExperiment& rhs);
   int            operator!=(const CMyExperiment& rhs);
public:
   // The member functions below allow us to override/extend base
   // class behavior for experiment specific stuff.

protected:
   virtual void SetupReadout(CExperiment& rExperiment);
   virtual void SetupScalers(CExperiment& rExperiment);

public:
   virtual void SetupRunVariables(CExperiment& rExperiment,
  CInterpreterStartup& rStartup,
  CInterpreterCore&    rCore);
   virtual void SetupStateVariables(CExperiment& rExperiment,
  CInterpreterStartup& rStartup,
  CInterpreterCore&    rCore);
   virtual void AddUserCommands(CExperiment& rExperiment,
  CInterpreterStartup& rStartup,
  CInterpreterCore&    rCore);

};

// The system relies on a globally accessible instance of a CReadoutMain
// derived object called MyApp so here it is:

CMyExperiment MyApp;

/*!
     In SetupReadout, you are expected to add Event segments to your
     experiment.   Event segments read out logical sections of your
     experiment.  The following types of event segments are available for your
     use:
       -    Simple event segments:  These are derived from the abstract base
              class CEventSegment, and are intended to readout a coherent piece
              of an experiment.
      -  Compatibility mode segments: These are objects of type
          CTraditionalEventSegment  they provide all the callouts to code
          that lives in an old-style skeleton.cpp file.  You will need to modify
          the skeleton makefile to add the oldstyle skeleton.cpp to the build.
          There can be only one CTraditionalEventSegment in the system due
           to function naming.
      -  Compound segments:  These are objects of type CCompoundEventSegment
          They consist of an ordered list of event segments of any type (including
          if you like other compound event segments.
```

```
       \param rExperiment - CExperiment& - A reference to the experiment object
                                                that runs the readout.  You will normall
              calls to CExperiment::AddEventSegment to register your
              own event segments in the experiment readout
*/
void
CMyExperiment::SetupReadout(CExperiment& rExperiment) // co:setupreadout
{
   CReadoutMain::SetupReadout(rExperiment);


   rExperiment.AddEventSegment(new MyEventSegment(10, 0xff00)); // co:addmysegment
                           // Make a new object of type MyEventSegment(slot#, ID)

}
/*!
    This function allows you to describe your scaler readout configuration.  This is done by
    inserting scalers into the experiment object.   Scalers come in the following flavors (a
    derived from CScaler
    - CCAMACScalerLRS2551  - CAMAC LeCroy model 2551 12 channel scalers.
    - CCAMACScalerLRS4434  - CAMAC LeCroy model 4434 32 channel scalers.
    - CVMEScalerCAENV830   - VME CAEN model V830 32 channel scalers.
    - CVMEScalerLRS1151         - VME LeCroy model 1151  scalers.
    - CScalerBank                         - A collection of scalers sequentially read out.

   \param rExperiment - CExperiment& reference to the experiment object.
    Normally  you will call CExperiment::AddScalerModule to add
    scaler modules to the readout.
*/
void
CMyExperiment::SetupScalers(CExperiment& rExperiment) // co:setupscalers
{
   CReadoutMain::SetupScalers(rExperiment);

   // Insert your code below this comment.

   // For test,setup an LRS 1151 at 0x200c00

   //   CScaler* pScaler = new CVMEScalerLRS1151(0xc00200);
   //   rExperiment.AddScalerModule(pScaler);

}
/*!
   In this function create and define any run variables you need.
   A run variable is a TCL Variable whose value is logged to the
   event stream.   Run variables are always modifiable.

   If, for example, you have a thermocouple that is monitoring
   the temperature of a temperature senstive detector, you could
   create a RunVariable, monitor the temperature periodically
   and update the RunVariable.  See CRunVariable and
   CRunVariableCommand  Run variables can also be
```

```
  create at the command line using the runvar command.

   \param rExperiment - CExperiment& the experiment object.
   \param rStartup   - CInterpreterStartup& the interpreter startup
                                            object.
    \param rCore  - CInterpreterCore& the core TCL interpreter
 add on functionality.  Normally you will obtain the run
 variable command object and do CRunVariableCommand::Create
 calls to add your run variables.

   \note The base class creates key run variables.  It is therefore
   very important to be sure the base class version of this function is
   called.
*/
void
CMyExperiment::SetupRunVariables(CExperiment& rExperiment,
  CInterpreterStartup& rStartup,
 CInterpreterCore&   rCore)
{
   CReadoutMain::SetupRunVariables(rExperiment, rStartup, rCore);

   CRunVariableCommand& rCommand(*(rCore.getRunVariables()));

   // Add your code below this commet. rCommand is a reference to the run variable
   // commands object.
}
/*
    This function allows you to create run state variables.  Run state
   variables are TCL variables that are write locked during a run.  Their
   values are logged to run state variable buffers at run state transitions.

   An example of a run state variable is the run number; created by the
   base class.  An example of run variables you might like to create are
   fixed run conditions, such as beam species, energy, target species,
   trigger conditions etc.

   The Tcl command statevar can also be used to create list and delete
   state variables.

   \param rExperiment - CExperiment& the experiment object.
   \param rStartup   - CInterpreterStartup& the interpreter startup
                                            object.
    \param rCore  - CInterpreterCore& the core TCL interpreter
 add on functionality.  Normally you will obtain the
 state variable command object and do CStateVariableCommand::Create
 calls to add your run variables.

   \note The base class creates key run variables.  It is therefore
   very important to be sure the base class version of this function is
   called.
*/
void
CMyExperiment::SetupStateVariables(CExperiment& rExperiment,
```

```
   CInterpreterStartup& rStartup,
   CInterpreterCore&    rCore)
{
   CReadoutMain::SetupStateVariables(rExperiment, rStartup, rCore);

   CStateVariableCommand& rCommand(*(rCore.getStateVariables()));

   // Insert your code below this comment.  Note that rCommand is the
   // state variable command object.

}
/*!
     Add user written commands in this function.  User written commands
   should be objects derived from CDAQTCLProcessor  This will ensure that
  command execution will be properly synchronized to the rest of the application.

   \param rExperiment - CExperiment& the experiment object.
   \param rStartup   - CInterpreterStartup& the interpreter startup
                                            object.  Normally you will use this object
   to locate the interpreter on which your commands
                                          will be registered.
    \param rCore  - CInterpreterCore& the core TCL interpreter
 add on functionality.

  \note The base class creates key command extensions (e.g. begin) it is
    important that the base class version of this function be called.

*/
void
CMyExperiment::AddUserCommands(CExperiment& rExperiment,
   CInterpreterStartup& rStartup,
   CInterpreterCore&    rCore)
{
   CReadoutMain::AddUserCommands(rExperiment, rStartup, rCore);

   CTCLInterpreter& rInterp(rStartup.Interp());

   // Add your command definitions after this comment.  rInterp
   // is a reference to the interpreter.
}
void* gpTCLApplication;
```

Example 5-4. Makefile

```
INSTDIR=/usr/opt/daq/8.1
# User makefile for the Production readout skeleton.  Read the comments
# in the code below to know what you can modify.
#
include $(INSTDIR)/etc/ProductionReadout_Makefile.include
```

```
#
#   Below, define any additional C++ compilation switches you might need.
#
#  If you are using camac you will need to add at least:
#    -DCESCAMAC
# or -DVC32CAMAC
# to select between the CES 8210 and WIENER CAMAC interface modules.
# respectively.
#
USERCXXFLAGS=
#
#  Below, define any additional C compilation switches you may need.
#  By default, this is defined to be the same as the C++ additional
#  switches:

USERCCFLAGS=$(USERCXXFLAGS)

#
#  Below, define any additional linker flags you may need:
#

USERLDFLAGS=

#
#  Add any objects you require to the defintion below. Note that the
#  Makefile knows by default how to correctly compile most C++ and C files.
#  Unless ther are problems, don't add any compilation rules for your objects.
#
#
#   If you are using a 'traditional readout skeleton,
#  Comment the Objects line two down and  uncomment the next line:
#
#Objects= Skeleton.o CTraditionalEventSegment.o CTraditionalScalerReadout.o
#                        co:listobjects
Objects=Skeleton.o MyEventSegment.o

#

Readout: $(Objects)
$(CXXLD) -o Readout  $(Objects) $(USERLDFLAGS) $(LDFLAGS)

clean:
rm -f $(Objects) Readout

depend:
makedepend $(USERCXXFLAGS) *.cpp

help:
echo "make       - Create the readout program."
echo "make clean - Clean up objects etc. created by previous builds"
echo "make depend- Add include file dependencies to the Makefile."
```

```
# DO NOT DELETE

MyEventSegment.o: MyEventSegment.h
Skeleton.o: MyEventSegment.h
```

Example 5-5. startreadout script

```
#!/bin/bash

. /etc/profile
. ~/.bashrc


if test "$DAQHOST" == ""
then
  host=‘hostname‘               # co:host
else
  host="$DAQHOST"
fi

/usr/opt/daq/8.1/bin/ReadoutShell -host=$host \
  -path=$HOME/experiment/readout/Readout # co:startrdo
```

5.2. SpecTcl software

Example 5-6. MyEventProcessor.h

```
#ifndef __MYEVENTPROCESSOR_H
#define __MYEVENTPROCESSOR_H

#include <EventProcessor.h>                     // co:epbaseclassinclude
#include <TranslatorPointer.h>                 // co:epxlatorpointerinclude
#include <histotypes.h>                         // co:histotypesinclude


// Forward class definitions:

class CTreeParameterArray;
class CAnalyzer;                  // co:epforwards
class CBufferDecoder;
class CEvent;
```

```
class MyEventProcessor : public CEventProcessor // co:epinherit
{
private:
  CTreeParameterArray&  m_rawData;              // co:treearraymember
  int                   m_nId;                  // co:epidmember
  int                   m_nSlot;                // co:epslotmember
public:
  MyEventProcessor(int         ourId,
   int         ourSlot,         // co:epconstructor
   const char* baseParameterName);
  ~MyEventProcessor();                    // co:epdestructor

  virtual Bool_t operator()(const Address_t pEvent,
    CEvent&         rEvent,   // co:epfunccall
    CAnalyzer&      rAnalyzer,
    CBufferDecoder& rDecoder);

private:
  void unpackPacket(TranslatorPointer<UShort_t> p);  // co:unpackUtil

};

#endif
```

Example 5-7. MyEventProcessor.cpp

```
#include <config.h>
#include "MyEventProcessor.h"            // co:epinclude
#include <TreeParameter.h>
#include <Analyzer.h>                    // co:epforwardincludes
#include <BufferDecoder.h>
#include <Event.h>

#include <iostream>

#include <TCLAnalyzer.h>                  // co:tclanalyzerinclude

#ifdef HAVE_STD_NAMESPACE
using namespace std;
#endif
static const ULong_t CAEN_DATUM_TYPE(0x07000000);
static const ULong_t CAEN_HEADER(0x02000000);
static const ULong_t CAEN_DATA(0);
static const ULong_t CAEN_TRAILER(0x04000000);
static const ULong_t CAEN_INVALID(0x06000000);
```

```
static const ULong_t CAEN_GEOMASK(0xf8000000);
static const ULong_t CAEN_GEOSHIFT(27);
static const ULong_t CAEN_COUNTMASK(0x3f00);   // co:constants
static const ULong_t CAEN_COUNTSHIFT(8);

static const ULong_t CAEN_CHANNELMASK(0x3f0000);
static const ULong_t CAEN_CHANNELSHIFT(16);
static const ULong_t CAEN_DATAMASK(0xfff);


static inline ULong_t getLong(TranslatorPointer<UShort_t>& p)  // co:getLong
{
  ULong_t high   = p[0];
  ULong_t low    = p[1];
  return (high << 16) | low;
}

MyEventProcessor::MyEventProcessor(int ourId,
   int ourSlot,
   const char* baseParameterName) :
  m_rawData(*(new CTreeParameterArray(string(baseParameterName),
      4096, 0.0, 4095.0, string("channels"),
      32, 0))),    // co:epinitializers
  m_nId(ourId),
  m_nSlot(ourSlot)
{
}



MyEventProcessor::~MyEventProcessor()
{
  delete &m_rawData; // co:epcleanup
}

Bool_t
MyEventProcessor::operator()(const Address_t pEvent,
     CEvent&         rEvent,
     CAnalyzer&      rAnalyzer,
     CBufferDecoder& rDecoder)
{
  TranslatorPointer<UShort_t> p(*(rDecoder.getBufferTranslator()),
pEvent);
  UShort_t                    nWords = *p++;           // co:epboilerplate
  CTclAnalyzer&               rAna((CTclAnalyzer&)rAnalyzer);
  rAna.SetEventSize(nWords*sizeof(UShort_t));
  nWords--; // Remaining words after word count.

  while (nWords) {
    UShort_t nPacketWords = *p;                       // co:packetHeader
    UShort_t nId          = p[1];
    if (nId == m_nId) {                               // co:packetSearch
      try {                                           // co:tryCatch
```

```
unpackPacket(p);                                // co:invokeUnpack
      }
      catch(string msg) {
cerr << "Error Unpacking packet " << hex << nId << dec
     << " " << msg << endl;
return kfFALSE;                                 // co:handleError
      }
      catch(...) {
cerr << "Some sort of exception caught unpacking packet " << hex << nId
     << dec << endl;
return kfFALSE;
      }
    }
    p      += nPacketWords;                          // co:nextPacket
    nWords -= nPacketWords;
  }
  return kfTRUE;                       // co:aOk
}

void
MyEventProcessor::unpackPacket(TranslatorPointer<UShort_t> pEvent)
{
  pEvent += 2;                     // co:toBody

  ULong_t header      = getLong(pEvent);
  if ((header & CAEN_DATUM_TYPE) != CAEN_HEADER) {   // co:headerSanity
    throw string("Did not find V785 header when expected");
  }
  if (((header & CAEN_GEOMASK) >> CAEN_GEOSHIFT) != m_nSlot) {
    throw string("Mismatch on slot");
  }

  Long_t nChannelCount = (header & CAEN_COUNTMASK) >> CAEN_COUNTSHIFT;

  pEvent += 2;                                       // co:toData
  for (ULong_t i =0; i < nChannelCount; i++) {       // co:decodeBody
    ULong_t channelData = getLong(pEvent);
    int     channel     = (channelData & CAEN_CHANNELMASK) >> CAEN_CHANNELSHIFT;
    int     data        = (channelData & CAEN_DATAMASK);
    m_rawData[channel] = data;

    pEvent += 2;
  }

  Long_t trailer = getLong(pEvent);
  trailer       &= CAEN_DATUM_TYPE;
  if ((trailer != CAEN_TRAILER) ) { // co:trailerSanity
    throw string("Did not find V785 trailer when expected");
  }

}
```

Example 5-8. MySpecTclApp.cpp

```
static const char* Copyright = "(C) Copyright Michigan State University 2008, All rights res

// Class: CMySpecTclApp

//////////////////////////// FILE_NAME.cpp /////////////////////////////////////////////////
#include <config.h>
#include "MySpecTclApp.h"
#include "EventProcessor.h"
#include "TCLAnalyzer.h"
#include <Event.h>
#include <TreeParameter.h>
#include "MyEventProcessor.h"

#ifdef HAVE_STD_NAMESPACE
using namespace std;
#endif


// Local Class definitions:


// This is a sample tree parameter event structure:
// It defines an array of 10  raw parameters that will
// be unpacked from the data and a weighted sum
// that will be computed.
//
typedef
struct  {
  CTreeParameterArray&  raw;
  CTreeParameter&       sum;
} MyEvent;

// Having created the struct we must make an instance
// that constructs the appropriate objects:


MyEvent event = {
  *(new CTreeParameterArray("event.raw", "channels",  10, 0)),
  *(new CTreeParameter("event.sum", "arbitrary"))
};

// Here's a sample tree variable structure
// that defines the weights for the weighted
// sum so that they can be varied from the command line:
// An array is also declared for testing purposes but not used.
typedef
struct {
  CTreeVariable&  w1;
```

```
  CTreeVariable&  w2;
  CTreeVariableArray& unused;
} MyParameters;

// Similarly, having declared the structure, we must define
// it and construct its elements

MyParameters vars = {
  *(new CTreeVariable("vars.w1", 1.0,  "arb/chan")),
  *(new CTreeVariable("vars.w2", 1.0, "arb/chan")),
  *(new CTreeVariableArray("vars.unused", 0.0, "furl/fort", 10, 0))

};
// CFixedEventUnpacker - Unpacks a fixed format event into
// a sequential set of parameters.
//

class CFixedEventUnpacker : public  CEventProcessor
{
public:
  virtual Bool_t operator()(const Address_t pEvent,
    CEvent&         rEvent,
    CAnalyzer&      rAnalyzer,
    CBufferDecoder& rDecoder);
};

Bool_t
CFixedEventUnpacker::operator()(const Address_t pEvent,
CEvent&         rEvent,
CAnalyzer&      rAnalyzer,
CBufferDecoder& rDecoder)
{

  // This sample unpacker unpacks a fixed length event which is
  // preceded by a word count.
  //
  TranslatorPointer<UShort_t> p(*(rDecoder.getBufferTranslator()), pEvent);
  CTclAnalyzer&      rAna((CTclAnalyzer&)rAnalyzer);
  UShort_t  nWords = *p++;
  Int_t     i      = 1;

  // At least one member of the pipeline must tell the analyzer how
  // many bytes were in the raw event so it knows where to find the
  // next event.

  rAna.SetEventSize(nWords*sizeof(UShort_t)); // Set event size.

  nWords--; // The word count is self inclusive.
  int param = 0; // No more than 10 parameters.

  while(nWords && (param < 10)) { // Put parameters in the event starting at 1.
    event.raw[param] = *p++;
    nWords--;
```

```
    param++;
  }
  return kfTRUE; // kfFALSE would abort pipeline.
}

// CAddFirst2 - Sample unpacker which adds a pair of unpacked parameters
//   together to get a new parameter.


class CAddFirst2 : public CEventProcessor
{
public:
  virtual Bool_t operator()(const Address_t pEvent,
    CEvent&         rEvent,
    CAnalyzer&      rAnalyzer,
    CBufferDecoder& rDecoder);

};

Bool_t
CAddFirst2::operator()(const Address_t pEvent,
       CEvent&         rEvent,
       CAnalyzer&      rAnalyzer,
       CBufferDecoder& rDecoder)
{
  event.sum = event.raw[0]*vars.w1 + event.raw[1]*vars.w2;
  return kfTRUE;
}


// Instantiate the unpackers we'll use.


static CFixedEventUnpacker Stage1;
static CAddFirst2          Stage2;

// CFortranUnpacker:
//   This sample unpacker is a bridge between the C++ SpecTcl
//   and a FORTRAN unpacking/analysis package.
//   The raw event is passed as a parameter to the FORTRAN function
//   f77unpacker.  This function has the signature:
//     LOGICAL F77UNPACKER(EVENT)
//     INTEGER*2 EVENT(*)
//
//  As with all event processors, this function is supposed to
//  return .TRUE. if processing continues or .FALSE. to abort event processing.
//  Unpacked events are returned to SpecTcl via a Common block declared
//  as follows:
//                                     ! generated by the unpacker:
//    COMMON/F77PARAMS/NOFFSET, NUSED, PARAMETERS(F77NPARAMS),
//   1                FPARAMETERS(F77NPARAMS)
//    INTEGER*4 PARAMETERS
//    LOGICAL   FPARAMETERS
```

```
//
//    At compile time, define: WITHF77UNPACKER
//    and F77NPARAMS to be the maximum number of parameters the Fortran
//    unpacker will fill in.
//
//    In the unpacker, set FPARAMETERS(i) if i has been unpacked from
//    the event.
//
//
//   PARAMETERS are copied from 1 -> NUSED into parameters numbered
//   NOFFSET -> NUSED+NOFFSET
//   NOFFSET starts from zero.
//
//   The fortran program should be compiled:
//      cpp -DF77NPARAMS=nnnn yourprog.f > yourprog.for
//      f77 -c yourprog.for
//      rm yourprog.for
//
//    Assuming you've writtne the file yourprog.f

#ifdef WITHF77UNPACKER


struct {
  int   nOffset;
  int   nUsed;
  int  nParameters[F77NPARAMS]; // Fortran will extend this appropriately.
  int fParameters[F77NPARAMS];
} f77params_;

extern "C" Bool_t f77unpacker_(const Address_t pEvent);

class CFortranUnpacker : public CEventProcessor
{
public:
  virtual Bool_t operator()(const Address_t pEvent, CEvent& rEvent,
    CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder);
};

Bool_t
CFortranUnpacker::operator()(const Address_t pEvent,
     CEvent&         rEvent,
     CAnalyzer&      rAnalyzer,
     CBufferDecoder& rDecoder)
{
  Bool_t result = f77unpacker_(pEvent);
  if(result) {
    int dest = f77params_.nOffset;
    for(int i = 0; i < f77params_.nUsed; i++) {
      if(f77params_.fParameters[i])
rEvent[dest] = f77params_.nParameters[i];
      dest++;
    }
```

```
    UShort_t* pSize = (UShort_t*)pEvent;
    CTclAnalyzer&  rAna((CTclAnalyzer&)rAnalyzer);
    rAna.SetEventSize( (*pSize)* sizeof(UShort_t));
  }
  return result;
}

CFortranUnpacker legacyunpacker;

#endif
//  Function:
//    void CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
//  Operation Type:
//     Override
/*
Purpose:

  Sets up an analysis pipeline.  This function is required and must
be filled in by the SpecTcl user.  Pipeline elements are objects
which are members of classes derived from the CEventProcessor
class. They should be added to the Analyzer's event processing pipeline
by calling RegisterEventProcessor (non virtual base class member).
   The sample implementation in this
file produces a two step pipeline.  The first step decodes a fixed length
event into the CEvent array.  The first parameter is put into index 1 and so on.
The second step produces a compiled pseudo parameter by adding event array
elements 1 and 2 and putting the result into element 0.


*/

void
CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
  RegisterEventProcessor(*(new MyEventProcessor(0xff00, 10,
"mydetectors.caenv785")));
}

// Constructors, destructors and other replacements for compiler cannonicals:

CMySpecTclApp::CMySpecTclApp ()
{
}

// Destructor:

 CMySpecTclApp::~CMySpecTclApp ( )
{
}


// Functions for class CMySpecTclApp
```

```
//  Function:
//    void BindTCLVariables(CTCLInterpreter& rInterp)
//  Operation Type:
//     override
/*
Purpose:

 Add code to this function to bind any TCL variable to
 the SpecTcl interpreter.  Note that at this time,
 variables have not yet been necessarily created so you
 can do Set but not necessarily Get operations.



*/
void
CMySpecTclApp::BindTCLVariables(CTCLInterpreter& rInterp)
{
  CTclGrammerApp::BindTCLVariables(rInterp);
}

//  Function:
//    void SourceLimitScripts(CTCLInterpreter& rInterpreter)
//  Operation Type:
//     Override
/*
Purpose:

Add code here to source additional variable setting
scripts.  At this time the entire SpecTcl/Tk infrastructure
is not yet set up.  Scripts run at this stage can only run
basic Tcl/Tk commands, and not SpecTcl extensions.
Typically, this might be used to set a bunch of initial values
for variables which were bound in BindTCLVariables.

*/
void
CMySpecTclApp::SourceLimitScripts(CTCLInterpreter& rInterpreter)
{ CTclGrammerApp::SourceLimitScripts(rInterpreter);
}

//  Function:
//    void SetLimits()
//  Operation Type:
//     overide
/*
Purpose:

Called after BindVariables and SourceLimitScripts.
This function can be used to fetch values of bound Tcl
variables which were modified/set by the limit scripts to
```

```
update program default values.

*/
void
CMySpecTclApp::SetLimits()
{ CTclGrammerApp::SetLimits();
}

//  Function:
//    void CreateHistogrammer()
//  Operation Type:
//     Override
/*
Purpose:

Creates the histogramming data sink.  If you want to override
this in general you probably won't make use of the actual
base class function.  You might, however extend this by
defining a base set of parameters and histograms from within
the program.

*/
void
CMySpecTclApp::CreateHistogrammer()
{ CTclGrammerApp::CreateHistogrammer();
}

//  Function:
//    void SelectDisplayer(UInt_t nDisplaySize, CHistogrammer& rHistogrammer)
//  Operation Type:
//     Override.
/*
Purpose:

Select a displayer object and link it to the
histogrammer.  The default code will link Xamine
to the displayer, and set up the Xamine event handler
to deal with gate objects accepted by Xamine interaction.

*/
void
CMySpecTclApp::SelectDisplayer(UInt_t nDisplaySize, CHistogrammer& rHistogrammer)
{ CTclGrammerApp::SelectDisplayer(nDisplaySize, rHistogrammer);
}

//  Function:
//    void SetupTestDataSource()
//  Operation Type:
//     Override
/*
Purpose:

 Allows you to set up a test data source.  At
```

```
present, SpecTcl must have a data source of some sort
connected to it... The default test data source produces a
fixed length event where all parameters are selected from
a gaussian distribution.  If you can figure out how to do it,
you can setup your own data source... as long as you don't
start analysis, the default one is harmless.

*/
void
CMySpecTclApp::SetupTestDataSource()
{ CTclGrammerApp::SetupTestDataSource();
}

//  Function:
//    void CreateAnalyzer(CEventSink* pSink)
//  Operation Type:
//     Override
/*
Purpose:

Creates an analyzer.  The Analyzer is connected to the data
source which supplies buffers.  Connected to the analyzer is a
buffer decoder and an event unpacker.  The event unpacker is
the main experiment dependent chunk of code, not the analyzer.
The analyzer constructed by the base class is a CTclAnalyzer instance.
This is an analyzer which maintains statistics about itself in Tcl Variables.

*/
void
CMySpecTclApp::CreateAnalyzer(CEventSink* pSink)
{ CTclGrammerApp::CreateAnalyzer(pSink);
}

//  Function:
//    void SelectDecoder(CAnalyzer& rAnalyzer)
//  Operation Type:
//     Override
/*
Purpose:

  Selects a decoder and attaches it to the analyzer.
A decoder is responsible for knowing the overall structure of
a buffer produced by a data analysis system.  The default code
constructs a CNSCLBufferDecoder object which knows the format
of NSCL buffers.

*/
void
CMySpecTclApp::SelectDecoder(CAnalyzer& rAnalyzer)
{ CTclGrammerApp::SelectDecoder(rAnalyzer);
}
```

```
//  Function:
//    void AddCommands(CTCLInterpreter& rInterp)
//  Operation Type:
//     Override
/*
Purpose:

This function adds commands to extend Tcl/Tk/SpecTcl.
The base class function registers the standard SpecTcl command
packages.  Your commands can be registered at this point.
Do not remove the sample code or the SpecTcl commands will
not get registered.

*/
void
CMySpecTclApp::AddCommands(CTCLInterpreter& rInterp)
{ CTclGrammerApp::AddCommands(rInterp);
}

//  Function:
//    void SetupRunControl()
//  Operation Type:
//     Override.
/*
Purpose:

  Sets up the Run control object.  The run control object
is responsible for interacting with the underlying operating system
and programming framework to route data from the data source to
the SpecTcl analyzer.  The base class object instantiates a
CTKRunControl object.  This object uses fd waiting within the
Tcl/TK event processing loop framework to dispatch buffers for
processing as they become available.

*/
void
CMySpecTclApp::SetupRunControl()
{ CTclGrammerApp::SetupRunControl();
}

//  Function:
//    void SourceFunctionalScripts(CTCLInterpreter& rInterp)
//  Operation Type:
//     Override
/*
Purpose:

  This function allows the user to source scripts
which have access to the full Tcl/Tk/SpecTcl
command set along with whatever extensions have been
added by the user in AddCommands.
```

```
*/
void
CMySpecTclApp::SourceFunctionalScripts(CTCLInterpreter& rInterp)
{ CTclGrammerApp::SourceFunctionalScripts(rInterp);
}

//  Function:
//    int operator()()
//  Operation Type:
//     Override.
/*
Purpose:

  Entered at Tcl/Tk initialization time (think of this
as the entry point of the SpecTcl program).  The base
class default  implementation calls the member functions
of this class in an appropriate order.  It's possible for the user
to extend this functionality by adding code to this function.

*/
int
CMySpecTclApp::operator()()
{
  return CTclGrammerApp::operator()();
}

CMySpecTclApp   myApp;
CTclGrammerApp& app(myApp); // Create an instance of me.
CTCLApplication* gpTCLApplication=&app;  // Findable by the Tcl/tk framework.
```

Example 5-9. Makefile

```
INSTDIR=/scratch/fox/SpecTcl/3.1test
# Skeleton makefile for 3.1

include $(INSTDIR)/etc/SpecTcl_Makefile.include

#  If you have any switches that need to be added to the default c++ compilation
# rules, add them to the definition below:

USERCXXFLAGS=

#  If you have any switches you need to add to the default c compilation rules,
#  add them to the defintion below:

USERCCFLAGS=$(USERCXXFLAGS)

#  If you have any switches you need to add to the link add them below:
```

```
USERLDFLAGS=

#
#   Append your objects to the definitions below:
#

OBJECTS=MySpecTclApp.o MyEventProcessor.o

#
#  Finally the makefile targets.
#


SpecTcl: $(OBJECTS)
$(CXXLD)  -o SpecTcl $(OBJECTS) $(USERLDFLAGS) \
$(LDFLAGS)


clean:
rm -f $(OBJECTS) SpecTcl

depend:
makedepend $(USERCXXFLAGS) *.cpp *.c

help:
echo "make                 - Build customized SpecTcl"
echo "make clean           - Remove objects from previous builds"
echo "make depend          - Add dependencies to the Makefile. "
```

Example 5-10. startspectcl

```
#!/bin/bash

. /etc/profile              # co:profile
. ~/.bashrc

cd ~/experiment/spectcl     # co:cd

./SpecTcl <setup.tcl        # co:spectclstart
```

Example 5-11. SpecTcl Setup file `setup.tcl`

```
source myspectra.tcl;     # co:definitions
sbind -all;               # co:bind

.gui.b update;            # co:treeupdate
```

```
if {[array names env DAQHOST] ne ""} {
    set daqsource $env(DAQHOST)
} else {;                   # co:daqsource
    set daqsource "localhost"
}

set url "tcp://$daqsource:2602"; # co:url

attach -pipe /usr/opt/daq/current/bin/spectcldaq $url; # co:attach
start;                                                 # co:start
```