Using NCSL DAQ Software to Readout a CAEN V775 TDC

Timothy Hoagland

November 12, 2004

Abstract

The paper's purpose is to assist the reader in setting up and reading out data from a CAEN V775. It covers what electronics will be needed and how they should be setup. It will show what modifications will need to be done to the software and gives code to be used. Testing for the code is also covered to a limited extent. Finally it covers how to get SpecTcl to produce a histogram of your events.

This paper assumes that you are at least somewhat familiar with Linux since the DAQ software runs on a Linux box. It also assumes that you a little familiar with C++. Finally it will be helpful if you know how to use an oscilloscope, as that will be needed to setup your electronics.

All the code is available at:
http://docs/daq/samples/CAEN V775/CAEN V775.zip

Contents

List of Figures

1 A Brief Description of the CAEN V775

The CAEN V775 is an 12-bit time digitizer. The output is a value proportional to the time between two logic pulses. It has 32 channels on a one unit wide VME module. The board can be operated in either common start or common stop mode. 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/

2 A Minimal Electronics Setup

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

- CAEN V775
- VME Crate
- VME controller
- NIM Crate
- NIM Discriminator
- NIM Gate and Delay generator
- NIM Delay Generator or very long cable
- NIM-ECL Converter
- VME CAEN V262 - I/O controller
- Pulser
- 50 Ohm LEMO terminator
- Various length LEMO cables
- Oscilloscope

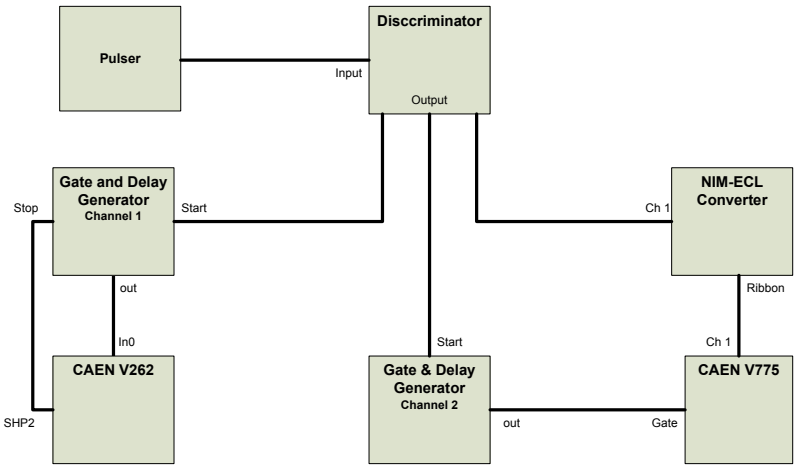

Figure 1: A simple electronics setup to readout a CAEN V775

Before starting make sure that you can secure all of these items, many are available through the NSCL electronics pool. Figure 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. More information about some of the modules we are going to use is available at http://docs/daq/samples/

We will first look at the signal from the pulser. A pocket pulser will work well for this. The signal directly from the pulser will look like figure 2 when viewed on a scope. The pulser only works when terminated with 50 Ohm.

The pulser will run directly into a NIM based discriminator. The discriminator puts out a logic signal every time the signal going into it reaches a certain threshold. While looking at both the input signal and the discriminator on the scope, adjust the threshold so that the discriminator only trips on the input signal. This means that you don't have to worry about noise in the channel triggering a logic signal. One of the discriminator outputs should go into the V775 via the NIM-ECL converter.

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

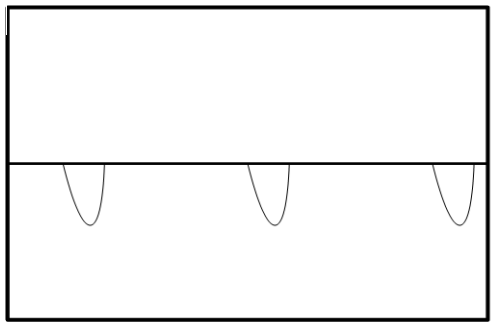

Figure 2: Pulser output signal

at the coloring on the cable you are using and be sure it is plugged in the correct channel of the TDC

One the discriminator outputs will need to be delayed to provide the V775 qith a stop signal. There are several ways to delay the second signal. Because the gate and delay generator I used had two channels I simply used one of those channels to delay the signal (if you do this be sure that the gate is set as narrow as possible). Another way of doing this would be to use cable to delay. Look at both the delayed and undelayed signal on the scope to be sure that the delayed signal comes about 400ns after the first pulse. When the timing is set, connect the delayed pulse into one of the common connectors on the V775, the other connector should be terminated with 50 Ohm.

The third 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 V775. 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 start signal is the signal from the discriminator. The discriminator output will go to the IN0 of the CAEN V262 I/O module. A cable running from the SHP2 output of the V262 to the stop of the gate and delay generator will deliver the computer generated stop.

At this point your setup should be complete. Now is good time to make sure that your setup is the same as Figure 1. If everything is setup correctly the BUSY and DRDY lights on the V775 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.

3 Sofware Modifications

The software modifications needed to make the above setup work can be divided into three tasks. First is telling the software what electronics we are using. Second is telling the software how to make sense of what it reads. Finally we have to tell the software how to graph what it has.

3.1 Readout Skeleton Modification

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/* .
```

3.1.1 Writing an Event Segment

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:

```
#ifndef __MYEVENTSEGMENT_H
#define __MTEVENTSEGMENT_H
#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
{
private:
    CDocumentedPacket m_MyPacket;
    CAENcard* module;
    unsigned short ID;
    short slot;
public:
  // Defines packet info
    MyEventSegment(short slot,unsigned short Id);
  // One time Module setup
    virtual void Initialize();
  // Resets data buffer
    virtual void Clear();
  // Reads data buffer
   virtual DAQWordBufferPtr& Read(DAQWordBufferPtr& rBuf);
   virtual unsigned int MaxSize();
};
#endif
```

This includes all of the definitions and classes that we will take advantage of, as well as declaring all of our functions. The next step is to implement the functions. A file called MyEventSegment.cpp should be created for that purpose. First we have to include the header file we just created, we will also

declare a packet version, which should be changed if major revisions are ever done to the event format.

```
#include "MyEventSegment.h"

static char* pPacketVersion = "1.0";
```

We can now declare our constructor. ID is the identification tag that will become associated with your packet. The parametes slot and Id will both be passed to the constructor from the Skeleton file.

```
MyEventSegment::MyEventSegment(short slot, unsigned short Id):
 m_MyPacket(Id, string("My Packet"),
 string("Sample documented packet"),
          string(pPacketVersion))
{
  // Creates a new CAENcard object looking
  //at the indicated VME
    module = new CAENcard(slot);
  slot
  // Sets the ID tag for the packet
    ID = Id;
 }
```

We now have to implement the four functions we declared in the header file, let's start with Initialize.

```
void MyEventSegment::Initialize()
{
  module->reset();  //reset defaults
  system("sleep .1s"); //pause while reset happens
  module->clearData(); //clear data buffer
  module->discardOverflowData();
  module->discardUnderThresholdData();
  module->commonStop();  //common stop mode
  module->setRange(0x1E); // Set full range
```

```
}
```

If you wanted to set a threshold, zero suppression or any other one-time setup detail, it would be included in the initialize function. The next function we will do is Clear.

```
void MyEventSegment::Clear()
{
    module ->clearData();
}
```

The MaxSize functions is no longer called but must still be defined we will define it as:

```
unsigned int MyEventSegment::MaxSize()
{
    return 32*4+2;
}
```

The Read function does two things, first it reads the data from the V775 and second it puts the data into a package for later analysis.

```
DAQWordBufferPtrt& MyEventSegment::Read(DAQWordBufferPtr& rBuf)
{
  for(int i=0; i < CAENTIMEOUT; i++)
  {
    if(module ->datapresent())
    {
      break;
    }
  }
  if(module -> dataPresent())
  {
```

```
    rBuf = m_MyPacket.Begin(rBuf);  //open a new packet
    module ->readEvent(rBuf);       //read data into packet
    rBuf = m_MyPacket.End(rBuf);    //close packet
  }
  return rBuf;
}
```

3.1.2 Modifying Skeleton.cpp

At this point we are now ready to modify the readout skeleton so it knows to access our new class. After opening Skeleton.cpp add:

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

Now locate the block of code that reads:

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

   // Insert your code below this comment.

   rExperiment.AddEventSegment(new MySegment);
}
```

Replace that block of code with:

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

```
   // Insert your code below this comment.

   rExperiment.AddEventSegment(new MyEventSegment(8,12));
}
```

The two numbers in the MyEventSegment call are the VME slot number and the packet ID, respectively. You will need to replace these numbers with values that reflect your own setup. Be sure to note what ID you use, that will be needed later.

3.1.3 Making Readout

We are now ready to compile our code but first we will have to modify the Makefile. Open Makefile and on the objects line add MyEventSegment.o. The line should look like this when your done.

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

Now simply type "make" at the command prompt. This will compile and link all the needed files and create a file called Readout. Once compilation errors have been fixed we are ready to test Readout.

3.1.4 Testing Readout

The first test is to see if the CAEN V775 is being read. After starting Readout type "begin". The small DTACK LED on the V775 should have come on. After you have confirmed that the DATCK light is on we can check that the data coming out of the TDC makes sense. We can do this by running a small program that dumps the data directly to the screen. In a second shell type:

```
/usr/opt/daq/Bin/bufdump
```

Data will start scrolling up the screen too fast to read. You can now go back to the first shell and type "end" to stop the readout. The data on the screen should look like:

```
-------------------- Event (first Event) ----------------
 Header:
4093 1 -12562 0 32 0 453 0 0 0 5 258 772 258 0 0
Event:
9(10) words of data
9 8 a 7200 100 7000 4779 7400
3cd2
```

This is a hexadecimal representation of the data being read off of the V775. The event that we made starts with 9, this is the total number of words (in hex) that exist in the event. The next word is the number of words in our packet (in hex). The third word is the packet id in this case 10. The remainder of the event is the data from the TDC event buffer. The next two words are the TDC header; if you break these down into binary representation the first five bits will be the slot where the V775 is located. If the packet ID and slot number are both correct then you are ready to move on to modifying the SpecTcl Skeleton.

3.2 Modifying SpecTcl

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/* .
```

3.2.1 Writing an Event Processor

We will create a derived class called MyEventProcessor to work with our readout program. The first step to that is to make another header file, this one will be called MyEventProcessor.h. It should look like:

```
#ifndef __MYEVENTPROCESSOR_H
```

```
#define __MYEVENTPROCESSOR_H
#include <EventProcessor.h>
#include <TranslatorPointer.h>
#include <BufferDecoder.h>
#include <Analyzer.h>
#include <TCLApplication.h>
#include <Event.h>

class MyEventProcessor : public CEventProcessor
{
private:
  // The Id Tag of the packets we want to unpack
    int nOurId;
  //The base element of the rbuf
  //array that we will store data in
    int Base;


public:
  //Class constrcutor
    MyEventProcessor(int ourId, int base);

  //Called to process each event
    virtual Bool_t operator()(const Address_t pEvent,
    CEvent& rEvent,CAnalyzer& rAnalyzer,
    CBufferDecoder& rDecoder);

protected:

  //unpacks and stores packet information in rBuf
    void UnpackPacket(TranslatorPointer<UShort_t>p,
    CEvent& rEvent);
};
#endif
```

The implementation file of should be called MyEventPrcossor.cpp. Start the file by including:

```
#include "MyEventProcessor.h"
#include <TCLAnalyzer.h>
#include <Event.h>
#include <FilterBufferDecoder.h>
```

We will begin by defining our constructor, which sets the value for our ID and our base index.

```
MyEventProcessor::MyEventProcessor(int ourId, int base)
                                        // Object Constructor
{
    nOurId =ourId; // Id tag that will be unpacking
    Base = base;   // starting index for storing
       //data in rbuf array
}
```

Now we can define the first function in our class. This function will read the number of words in the event then search through those words looking for our ID tag, if it finds our packet it orders it unpacked.

```
Bool_t
MyEventProcessor::operator()(const Address_t pEvent,
CEvent& rEvent, CAnalyzer& rAnalyzer,
CBufferDecoder& rDecoder)
{
  TranslatorPointer<UShort_t>
      p(*(rDecoder.getBufferTranslator()),
      pEvent);
  CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);

  UShort_t nWords = *p++; // Word count.
  rAna.SetEventSize(nWords*sizeof(UShort_t));
  nWords--; // past our event
```

```
  while(nWords)  // search for nOurID
  {
    UShort_t nPacketWords = *p;
    UShort_t nId    =  p[1];
    if(nId == nOurId)  // found our packet
      {
      UnpackPacket(p, rEvent);
      }
    p      += nPacketWords;   // book keeping to keep
    nWords -= nPacketWords; // word count right
  }
  return kfTRUE;
}
```

The second function is the function that unpacks our packet and stores it on the rBuf array.

```
void
MyEventProcessor::UnpackPacket(TranslatorPointer<UShort_t>
                pEvent, CEvent& rEvent)
{
  UInt_t nPacketSize = *pEvent;
  ++pEvent; ++pEvent; ++pEvent; ++pEvent;
  // Skip header of packet.
  nPacketSize -= 4;  // Remaining number of words.

  Int_t nchannel // TDC channel data is from

  while(nPacketSize >2) // ignore end of block
  {
    // read the first data word
    UInt_t word1 = *pEvent;
    // get last five bits from word1 to get channel number
    nchannel = word1 & 0x1f;

    ++pEvent;        //advance read pointer
```

```
    UInt_t word2 = *pEvent;  // read the second data word
    // get last twelve bits of word 2 = digitized value
    UInt_t TDCValue = word2 & 0x7ff;
    // place the TDCValue into the [th] element of rbuf array
    rEvent[Base + nchannel] = TDCValue;

    ++pEvent;        // advance read pointer
    nPacketSize -=2;         // adjust remaining word count


    //next line is a test line to be removed later
    cerr<< "found a value of " << TDCValue
        <<" at channel " << nchannel <<endl;
    // test line - remove after testing
  }
  return rBuf;
}
```

3.2.2 Modifying MySpecTclApp.cpp

We are now ready to make some changes to the MySpecTclApp.cpp file. After the last #include add these lines:

```
#include MyEventProcessor.h
MyEventProcessor MyProcessor;
```

Now locate the CreateAnalysisPipeline function that looks like this:

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

#ifdef WITHF77UNPACKER
  RegisterEventProcessor(legacyunpacker);
#endif
```

```
    RegisterEventProcessor(Stage1);
    RegisterEventProcessor(Stage2);
}
```

Edit the function so that it uses the MyEventProcessor class we just made. The values of our base index and our and our packet Id should be passed here. The base index is the starting array index that the values will be stored at, the Id is the same as you used earlier.

```
void
CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
RegisterEventProcessor(MyProcessor(101, 12);
}
```

3.2.3 Making SpecTcl

We are now ready to edit our Makefile so that we can compile our program. The first task is to find the line that reads:

```
OBJECTS=MySpecTclApp.o
```

and make it read

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

The second change is to add a couple of lines to the end of the file.

```
MyEventProcessor.o: MyEventProcessor.cpp MyEventProcessor.h
        $(CXXCOMPILE) MyEventProcessor.cpp
```

You can now run "make" to compile your code to make a file called SpecTcl. Fix all compilation errors before continuing.

3.3 Writing the SpecTecl Script

The last piece of code we have to write is the spectcl setup script. This the piece of code that tells the software what is available to histogram. Start by creating a file called setup.tcl that looks like

```
set slot 101;    #TDCbase address used earlier earlier

#Define 32 parameters named tdc0...tdc31 starting in slot 101
for {set i 0} {$i <= 31} {incr i} {
   parameter tdc$i $slot  12;
   incr slot
}

#define a 1-d spectrum for each parameter:
for {set i 0} {$i <= 31}  {incr i} {
    spectrum tdc$i 1 tdc$i {{0 4095 2048}; # 12 bit
}

sbind -all;   #make all spectrum displayable
```

3.4 Attaching to live data

To process data online as you will need to add an "Attach online" button that will get the data being streamed from the readout program. This script will be called "AttachButton" and will include a call to our setup.tcl script

```
source setup.tcl
proc Attach {}  {
   attach -pipe /usr/opt/daq/Bin/spectcldaq tcp://spdaqXX:2602/
   start
}
button .attach -command Attach -text "Attach Online"
pack    .attach
```

The spdaqXX above must be replaced with the DAQ computer you are working at.

4 Testing and Running

We are now ready to test our event processor. Begin by starting Readout and beginning a run. Then start SpecTcl.

4.1 Testing SpecTcl

When SpecTcl starts a window with a command prompt will open type:

```
source AttachButton
```

This will create an "attach online" button in the SpecTcl window. Click this attach online button. The window in which you started SpecTcl should now be scrolling text. This is the test line we added in our code earlier. If you now go to the window where you are running readout and type "end" the scrolling will stop and you can see what you have. The last line should look like:

```
found a value of 1916 at channel 0
```

The value found should be in the channel your signal is going into, in this case channel 0. If your readout does not look like this go back and look at the your code. Otherwise remove the test line we included in MyEventProcessor.cpp and make your file again.

4.2 Testing everything together

After removing the test line repeat, the last test. This time you should have no text scrolling. Follow these steps to see your spectrum:

- Click on the Xamine window

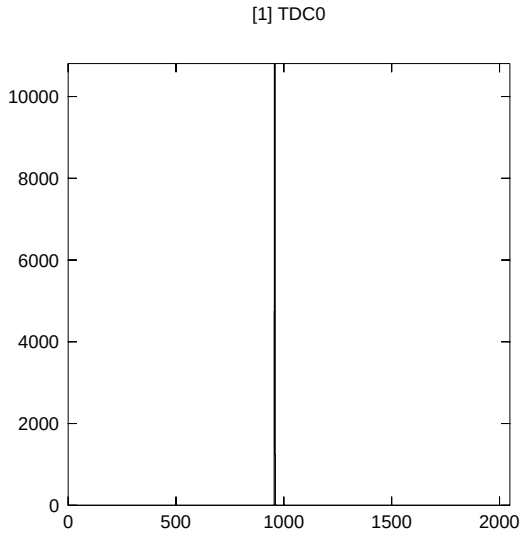

Figure 3: A sprectrum of Channel 1 from the CAEN V775 TDC

- Click on the Display button
- Select the TDC channel you are using.
- Click Ok

You should now have a histogram of your TDC channel on the screen that looks similar to figure 6. Now display a channel that you aren't using to be sure that it is different, it should be empty.

Now change what channel your pulser is going into, this will ensure that you can read other channels as well.

5 More information

You have now developed a fully functional DAQ system. Although it is not very complicated it can be very versatile. With only small changes you will be able read a CAEN V785, or V792. Slightly more changes are required to read out other modules but the steps and ideas that are in MyEventSegement and MyEventProcessor are the same regardless of what modules you are reading.

More information is available at: **http://docs.nscl.msu.edu/** and should be your first source for help. If that doesn't help contact **daq@nscl.msu.edu**

Please help with the accuracy of the paper. If you find any kind of error please report it at **daq@nscl.msu.edu**.

6 Complete Sample Code

The complete code needed to readout out a V792 can be found at http://docs.nscl.msu.edu/daq/samples/. The computer you are using will need to be specified in the "abutton" script and both the Readout code and SpecTcl code will need be compiled.

6.1 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
#define __MTEVENTSEGMENT_H
#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
{
private:
  CDocumentedPacket m_MyPacket;
  CAENcard* module;
  unsigned short ID;
  short slot;
public:
  MyEventSegment(short slot,unsigned short Id);
              // Defines packet info

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

  virtual void Clear();
            // Resets data buffer

  virtual unsigned int MaxSize();

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

6.2 MyEventSegment.cpp

```
/*
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 "MyEventSegment.h"

static char* pPacketVersion = "1.0";
// Packet version -should be changed whenever major changes are made

//constructor set Packet details
MyEventSegment::MyEventSegment(short slot, unsigned short Id):
  m_MyPacket(Id, string("My Packet"), string("Sample documented packet"),
             string(pPacketVersion))
{
  // Creates a new CAENcard object looking at the indicated VME slot
  module = new CAENcard(slot);

  // Sets the ID tag for the packet
  ID = Id;
}


// Is called right after the module is created.  All one time Setup
// should be done now.
void MyEventSegment::Initialize()
{
  module->reset();  //reset defaults
  system("sleep .1s"); //pause while reset happens
  module->clearData(); //clear data buffer
  module->discardOverflowData();
  module->discardUnderThresholdData();
  module->commonStop();  //common stop mode
  module->setRange(0x1E); // Set full range
}


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

unsigned int MyEventSegment::MaxSize()
{
  return 32*4+2;
}
```

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

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

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

6.3 MyEventProcessor.h

```
/*
This is the header file for the MyEventProcessor
class.  This class defines functions that checks the
packet ID of a packet and unpacks it if the ID is
recongnized. UnpackPacket is good for any CAEN
module covered by the CAENcard class.

Tim Hoagland
11/3/04
s04.thoagland@wittenberg.edu
*/
#ifndef __MYEVENTPROCESSOR_H
#define __MYEVENTPROCESSOR_H
#include <EventProcessor.h>
#include <TranslatorPointer.h>
#include <BufferDecoder.h>
#include <Analyzer.h>
#include <TCLApplication.h>
#include <Event.h>

class MyEventProcessor : public CEventProcessor
{
 private:
  int nOurId;   // The Id Tag of the packets we want to unpack
  int Base;     // The base element of the rbuf array that we will store data in

 public:
  // Class constrcutor
  MyEventProcessor(int ourId, int base);

  // Tests if a given packets Id = nOurId
  virtual Bool_t operator()(const Address_t pEvent, CEvent& rEvent,
                            CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder);
 protected:
  // unpacks and stores packet information in rBuf
  void UnpackPacket(TranslatorPointer<UShort_t>p, CEvent& rEvent);
};
#endif
```

6.4 MyEventProcessor.cpp

```
/*
This is the implementation file for the MyEventProcessor
class.  This class defines functions that checks the
packet ID of a packet and unpacks it if the ID is
recongnized. UnpackPacket is good for any CAEN
module covered by the CAENcard class.

Tim Hoagland
11/3/04
s04.thoagland@wittenberg.edu
*/
#include "MyEventProcessor.h"
#include <TCLAnalyzer.h>
#include <Event.h>
#include <FilterBufferDecoder.h>
int i=0;

MyEventProcessor::MyEventProcessor(int ourId, int base) // Object Constructor
{
  nOurId =ourId;        // Id tag that will be unpacking
  Base = base;          // starting index for storing data in rbuf array
}

Bool_t MyEventProcessor::operator()(const Address_t pEvent, CEvent& rEvent,
                                    CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder)
{
  TranslatorPointer<UShort_t> p(*(rDecoder.getBufferTranslator()), pEvent);
  CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);
  UShort_t nWords=*p++;           // get number of words in the packet
  rAna.SetEventSize(nWords*sizeof(UShort_t));
  nWords--;

  while(nWords)                  // Search packet for ID tag
    {
      UShort_t nPacketWords=*p;
      UShort_t nId = p[1];
      if(nId == nOurId)
        {
          UnpackPacket(p, rEvent);
        }
      p +=nPacketWords;
      nWords -= nPacketWords;
    }
  return kfTRUE;
}


void MyEventProcessor::UnpackPacket(TranslatorPointer<UShort_t> pEvent,
                                    CEvent& rEvent)
{
  UInt_t nPacketSize = *pEvent;  // get number of words let to be read
  ++pEvent;++pEvent;++pEvent;++pEvent; // skip over Id info and Header words
  nPacketSize -=4;               // adjust words remaining
```

```
  Int_t nchannel;
  while(nPacketSize>2)  // ignore end of block words
    {
      UInt_t word1 = *pEvent;   // read the first data word
      nchannel = word1 & 0x1f;  // get last five bits from word1 to get channel number
      ++pEvent;                 //advance read pionter
      UInt_t word2 = *pEvent;   // read the second data word
      UInt_t TDCValue = word2 & 0x7ff;  // get last twelve bits of word 2 = digitized value
      rEvent[Base + nchannel] = TDCValue;  // place the TDCValue into the [th] element of rbuf array
      ++pEvent;                 // advance read pionter
      nPacketSize -=2;          // adjust remaing word count

      // Below this line is a test line that can be uncommented to look at raw data
      //cerr << "found a value of " <<TDCValue << " at channel " << nchannel <<endl;
    }
return rEvent;
}
```