Using NCSL DAQ Software to Readout a CAEN V1x90 Multi-Hit TDC

Timothy Hoagland

January 7, 2005

Abstract

This paper's purpose is to assist the reader in setting up and reading out data from a CAEN V1190 or V1290. It covers what electronics will be needed and how they should be setup. It will show what software modifications need to be done and gives sample code to use.

Since this module is very complicated someone with no experience may find it useful to read through information on setting up simpler modules such as the V775 first. 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 V1290/CAEN V1290.zip

Much of the code and information in this guide was written by Ron Fox and taken, either verbatim or in spirit, from
http://docs.nscl.msu.edu/daq/modules/

Contents

1 A Brief Description of the CAEN V1290 **3**

2 Operation modes of the V1290 **3**
2.1 Trigger and Clock Timing 3
2.2 Trigger Matching Mode 4
2.2.1 Trigger Matching Window Timing 4
2.3 Continuous Storage mode 6

3 Setting up for Common Stop Mode **6**
3.1 Readout code for the Common Stop Mode 6
3.1.1 MyEventSegment.h 8
3.1.2 Some functions from CCAENV1x90 8
3.1.3 MyEventSegment.cpp 10
3.1.4 Compiling and Testing 11
3.2 Making SpecTcl 12
3.2.1 CCAENV1x90Data 12
3.2.2 V1x90Processor.h 12
3.2.3 V1x90Processor.cpp 13
3.2.4 A closer look at C1x90Processor::operator 15
3.2.5 Modifying MySpecTclApp.cpp 17
3.2.6 Writing setup.tcl 17
3.2.7 Compiling 18

4 Notes about common start **18**

5 More Information **18**

List of Figures

1 Relative trigger window timing 5
2 An example of how triggers are processed. 5
3 Window Timing for a V1290 in "Common Stop Mode" 6
4 A simple Electronics setup to use a V1290 as a TDC in "Common Stop Mode" 7
5 Simple electronics setup for running a V1290 in "Common Start" mode. 19

1 A Brief Description of the CAEN V1290

The CAEN V1190 and V1290 are multihit TDCs. Both modules take advantage of the same 32 channel CERN/ECP-MIC HPTDC chip, which provides up to 100ps time resolution. The V1290 staggers the start of four chip channels to produce a module with up to 25ps resolution. Both have two modes of operation, continuous storage and trigger matching. In trigger matching mode the module measures the time between a signal and the master trigger, many signals can be measured with respect to one master trigger. In continuous storage mode the module records a time for each signal received with respect to the last reset. 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/. NSCL support for this module is provided in the form of two C++ classes, CCAENV1x90 and CCAENV1x90Data.

NOTE: The remainder of this tutorial will refer only to the V1290, since at the time of writing it was the only one in service at the NSCL. Please keep in mind that everything done here will also work for the V1190.

2 Operation modes of the V1290

The V1290 is a very complicated module that requires more knowledge to use than many other VME modules, to that end it is important to spend some time talking about the module before starting to use it.

2.1 Trigger and Clock Timing

One of the easiest traps to fall into using the V1290 is to use the trigger time as a reference time. In traditional common start/stop TDCs the trigger timing is used to start/stop the digitization of all the channels on the module. The V1290 cannot be used like this because the trigger-timing stamp comes from the module clock, which cycles every 25ns. That means, even though the channel signal was measured with high precision the trigger will only be measured to 25ns. The work around for this problem is to send the trigger signal into channel 0 of the module and use it as the time reference instead of the trigger. More detail on this given later along with an example

2.2 Trigger Matching Mode

Most groups at the NSCL will want to use the V1290 as a traditional TDC, using it because of the high channel density or time resolution. By running in trigger matching mode the V1290 can be run as a common start or stop TDC with a range of 52 microseconds. So that it is possible to understand the initializations of the module in either mode it is important to look closely at the timing of the trigger matching window.

2.2.1 Trigger Matching Window Timing

There are four programmable parameters associated with the timing of the trigger matching window; window offset, reject margin, extra search margin, and match window width. The reject margin is the time before the start of the match window during which any triggers are ignored. The window offset is the offset timing from the trigger input and the start of the matching window, if this is set to a negative value the matching window will start before the trigger input arrives, if it is a positive value the matching window starts after the trigger input. The matching window width is exactly what you might think it is, the time during which the module will accept triggers.

The last parameter, the extra search margin, requires a little more understanding of the module. The V1290 has multiple levels of memory storage, including a L1 buffer and the readout buffer. When data is transferred from the level 1 buffer to the readout buffer the hits may not be recorded in the order in which they arrived, especially if a lot of hits arrive simultaneously. The extra search margin allows the module to continue transfer data from the L1 buffer to the readout buffer until it finds a hit that occurred after the extra search margin. A graphical representation of all four parameters is in Figure 1.

To better understand how all of the parameters work with incoming triggers we will look at the example shown in figure 2. Hit 0 occurs within the reject margin and will be completely ignored by the module. Hits 1-7 all occur during the matching window. They will be measured with respect to the trigger. Hit 8 and 9 occur within the extra search margin. Both hits will be counted since they arrived within the search area and will be treated as though they had occurred within the matching window.

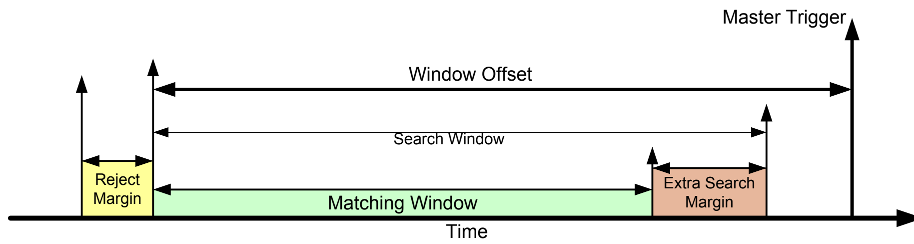

Figure 1: Relative trigger window timing

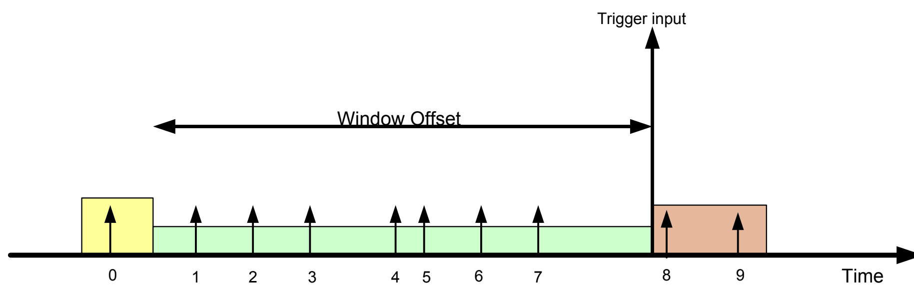

Figure 2: An example of how triggers are processed.

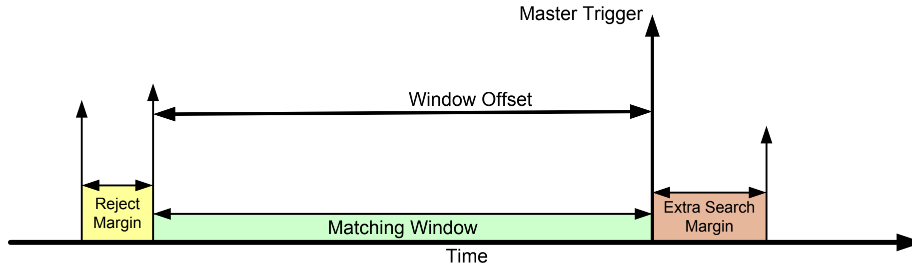

Figure 3: Window Timing for a V1290 in "Common Stop Mode"

2.3 Continuous Storage mode

When running the V1290 in continuous storage mode each hit is given a time stamp with its arrival time with respect to the last reset. The module can accept hits for 52 microseconds before it needs to be retriggered. Double hit resolution can be set as low as 5ns. No example of using the module in this mode will be given.

3 Setting up for Common Stop Mode

To use the module as a regular TDC in common start mode the module should be run in trigger matching mode. The windowing parameters should be set as seen in figure 3. The Electronics setup would look like figure 4. The trigger signal is put into both the trigger and ch0 to compensate for the 25ns trigger resolution issues discussed above. By recording the trigger timing with a channel of the TDC, we can get an accurate and precise trigger timing that can then be subtracted from for the start signal going into channel 1 to get a high resolution time measurement.

3.1 Readout code for the Common Stop Mode

To make the readout program for the V1290 we will create a C++ class called MyEventSegment.

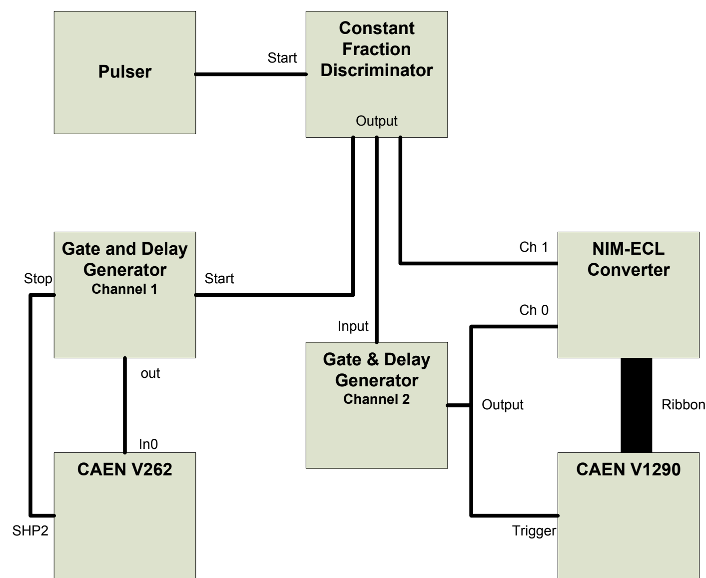

Figure 4: A simple Electronics setup to use a V1290 as a TDC in "Common Stop Mode"

3.1.1 MyEventSegment.h

The header file for the class will look like:

```
#ifndef __MYEVENTSEGMENT_H
#define __MTEVENTSEGMENT_H
#include <spectrodaq.h>
#include <CEventSegment.h>
#include <CDocumentedPacket.h>
#include  <CCAENV1x90.h>


// Declares a class derived from CEventSegment
class MyEventSegment : public CEventSegment
{
private:
  CDocumentedPacket m_MyPacket;
  CCAENV1x90* m_TDC;
  unsigned int Slot;
  unsigned long Base;
  unsigned int Crate;
public:
// Defines packet info
 MyEventSegment(unsigned int slot,
               unsigned int crate,
               unsigned long Base);

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

// Resets data buffer
  virtual void Clear();

  virtual unsigned int MaxSize();

// Reads data buffer
  virtual DAQWordBufferPtr& Read(DAQWordBufferPtr& rBuf);

};
#endif
```

3.1.2 Some functions from CCAENV1x90

It is important to note that the header file includes ”CCAENv1x90.h”, this is the class that provides support for the module. Many of the functions defined within that class will be used in the implementation of MyEventSegment::Initialization. Some of the functions that should be mentioned are:

- CCAENV1x90::TriggerMatchMode - turns on trigger match mode.
- CCAENV1x90::SetWindowWidth - Sets the width of the matching window.

- CCAENV1x90::SetWindowOffset - Sets the position of the start of the window relative to the trigger time.
- CCAENV1x90::SetRejectMargin - Sets the initial rejection margin.
- CCAENV1x90::SetExtraSearchMargin - sets the extra search margin.
- CCAENV1x90::EnableTriggerTimeSubtraction - ensures that the hit times will be relative to the match window start.
- CCAENV1x90::SetEdgeDetectMode - To set the module for leading edge detection.
- CCAENV1x90::SetIndividualLSB - To set the meaning of the Least significant bit of the TDC.
- CCAENV1x90::SetDoubleHitResolution - to set the double hit resolution.
- CCAENV1x90::EnableTDCEncapsulation - to enable the hits from a single chip to be encapsulated in chip header/trailers.
- CCAENV1x90::SetMaxHitsPerEvent - to set the number of hits we allow each chip to record.
- CCAENV1x90::EnableErrorMark - to enable the production of error information in the event buffer.
- CCAENV1x90::SetErrorEnables - to determine the set of errors we want the chip to produce.
- CCAENV1x90::EnableAllChannels - to enable all channels in the module.

Complete documentation about the CCAENV1x90 class can be found at http://docs.nscl.msu.edu/daq/modules/

3.1.3 MyEventSegment.cpp

The implementation file for the MyEventSegment class should look like:

```
#include "MyEventSegment.h"

static char* pPacketVersion = "1.0";
// Packet version -should be changed whenever
major changes are made
int Id=12;


//constructor set Packet details
MyEventSegment::MyEventSegment(unsigned int slot,
                               unsigned int crate,
                               unsigned long  base):
  m_MyPacket(Id, string("My Packet"),
             string("Sample documented packet"),
       string(pPacketVersion))
{
  m_TDC = new CCAENV1x90(slot, crate, base);
}

void MyEventSegment::Initialize()
{
// Set the trigger matching mode and associated parameters:

   m_TDC->TriggerMatchMode();
   m_TDC->SetWindowWidth(3*40);
   m_TDC->SetWindowOffset(-2*40);
   m_TDC->SetExtraSearchMargin(1);
   m_TDC->SetRejectMargin(1);
   m_TDC->EnableTriggerTimeSubtraction();

      // Set detection and resolution stuff:

   m_TDC->SetEdgeDetectMode(CCAENV1x90::EdgeMode_Leading);
   m_TDC->SetIndividualLSB(CCAENV1x90::Res_100ps);
   m_TDC->SetDoubleHitResolution(CCAENV1x90::DT_5ns);

     // Manage the event format:

   m_TDC->EnableTDCEncapsulation();
   m_TDC->SetMaxHitsPerEvent(CCAENV1x90::HITS_UNLIMITED);
   m_TDC->EnableErrorMark();
   m_TDC->SetErrorEnables(CCAENV1x90::ERR_VERNIER     |
                          CCAENV1x90::ERR_COARSE      |
                          CCAENV1x90::ERR_SELECT      |
                          CCAENV1x90::ERR_L1PARITY    |
                          CCAENV1x90::ERR_TFIFOPARITY |
                          CCAENV1x90::ERR_MATCHERROR  |
                          CCAENV1x90::ERR_RFIFOPARITY |
                          CCAENV1x90::ERR_RDOSTATE    |
                          CCAENV1x90::ERR_SUPPARITY   |
                          CCAENV1x90::ERR_CTLPARITY   |
                          CCAENV1x90::ERR_JTAGPARITY);
   m_TDC->EnableAllChannels();
```

```
}

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

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


#define TDC_WAITLOOP 5
static unsigned long TDCData[32768];

//Is called to readout data on module
DAQWordBufferPtr& MyEventSegment::Read(DAQWordBufferPtr& rBuf)
{
  for(int i =0; i < TDC_WAITLOOP; i++ ) {
     if(m_TDC->isEventFIFOReady()) {
        break;
     }
  }
  if(m_TDC->isEventFIFOReady()) {
     unsigned long fifo = m_TDC->ReadEventFIFO();
     unsigned nWords    = m_TDC->FIFOWordCount(fifo);

     unsigned nRead     = m_TDC->ReadData(TDCData, nWords);
     rBuf.CopyIn(TDCData, 0, nRead*2);
     rBuf += nRead*2;
  }

  return rBuf;
}
```

3.1.4 Compiling and Testing

After your implementation file is complete the last thing you need to do is to tie MyEventSegment into the Readout Skeleton. To do this define a module in the CMyExperiment::SetupReadout function, by adding a line such as:

```
rExperiment.AddEventSegment(new MyEventSegment(4, 0, 0x1000000));
```

Where the first argument is the VME slot number the module is in, the second is the VME crate number, and the last argument is the module's user defined base address. You will also need to add this line to the list of includes:

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

After you have added these lines into Skeleton.cpp you are ready to compile Readout. At this point you should start a readout run and run bufdump to make sure that the module is being read.

3.2 Making SpecTcl

After you have Readout running successfully you can move on to making SpecTcl. This will require writing a C++ class called V1290Processor. Unlike other modules there is a special class used to unpack and make sense of the output buffer of the V1290. That class is called CCCAENV1x90Data.

3.2.1 CCAENV1x90Data

Because the V1290 is a multihit TDC the number of words and the order of the words in each event are not also easy to predict, for that reason each word in the event will have to be looked at to determine what kind of data it holds, after it is determined what a word contains appropriate action has to be taken to extract the information we need from that word. To make this easier we will take advantage of the aforementioned CCAENV1x90Data class. Complete documentation on this class is available at http://docs.nscl.msu.edu/daq/modules/.

3.2.2 V1x90Processor.h

The header file for V1x90processor 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>
#include <CCAENV1x90Data.h>

class MyEventProcessor : public CEventProcessor
{
 private:
  unsigned int m_nVsn;
  unsigned int m_nBaseParameter;

 public:

  // Class constrcutor
```

```
  MyEventProcessor(unsigned int nSlot,
                   unsigned int nBase);

  virtual Bool_t operator()(const Address_t pEvent,
                            CEvent& rEvent,
                            CAnalyzer& rAnalyzer,
                            CBufferDecoder& rDecoder);
};
#endif
```

3.2.3 V1x90Processor.cpp

The implementation file for V1x90Processor will look like this:

```
#include "MyEventProcessor.h"
#include <TCLAnalyzer.h>
#include <Event.h>
#include <FilterBufferDecoder.h>
#define MAXHITSPERCHANNEL 1

int i=0;

MyEventProcessor::MyEventProcessor(unsigned int nSlot,
                                   unsigned int nBase) :
  m_nVsn(nSlot),
  m_nBaseParameter(nBase)
{

}

Bool_t MyEventProcessor::operator()(void* pEvent,
                                    CEvent& rEvent,
            CAnalyzer& rAnalyzer,
                                    CBufferDecoder& rDecoder)
{
  TranslatorPointer<UShort_t>
              pw(*(rDecoder.getBufferTranslator()), pEvent);
  UShort_t nWords = *pw++;
  CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);
  rAna.SetEventSize(nWords*sizeof(UShort_t));
  ULong_t nLongs = (nWords-1)*sizeof(UShort_t)/sizeof(ULong_t);

  int nSlot  = -1;
  bool FormatOK;
  long hits[32][MAXHITSPERCHANNEL];
  long nHits[32];
  for (int i =0; i < 32; i++) {
    nHits[i] =0;}


  while(nLongs)                  // Search packet for ID tag
    {
      ULong_t data = *pw;
      ++pw;
```

```
      data |= *pw<<16;
      ++pw;

      nLongs--;
      if(CCAENV1x90Data::isGlobalHeader(data)) {
nSlot = CCAENV1x90Data::BoardNumber(data);
      }
      if(CCAENV1x90Data::isTDCHeader(data)) {
      }
      if(CCAENV1x90Data::isTDCTrailer(data)) {
      }
      if(CCAENV1x90Data::isTDCError(data)) {
      }
      if(CCAENV1x90Data::isMeasurement(data)) {
int nChannel = CCAENV1x90Data::ChannelNumber(data, false);
long nValue  = CCAENV1x90Data::ChannelValue(data, false);
if(nSlot == m_nVsn) {
  if(nHits[nChannel] < MAXHITSPERCHANNEL) {
    hits[nChannel][nHits[nChannel]] = nValue;
    nHits[nChannel]++;
  }
}
      }
      if(CCAENV1x90Data::isGlobalTrailer(data)) {
if(nLongs) {
  FormatOK = false;
} else {
  FormatOK = (nSlot == m_nVsn);
}
      }
    }

  if(FormatOK) {
    if(nHits[0] == 1)
      {
      long nGateTime = hits[0][0]; // Gate time.
      int nParam = m_nBaseParameter;
      for(int nChan = 1; nChan < 2; nChan++) {
        for(int ihit = 0; ihit < nHits[nChan]; ihit++) {
          rEvent[nParam + ihit] = nGateTime - hits[nChan][ihit];
        }
        nParam += MAXHITSPERCHANNEL;

        }
      }

  }
      return kfTRUE;
}
```

3.2.4 A closer look at C1x90Processor::operator

The function that is responsible for making sense of the data from the output buffer is C1x90processor::operator. Since this is a very complicated function it is worth spending some time looking more closely at how it works.

This line creates a translation buffer to ensure the internal byte ordering of each word is understood regardless of what type of machine you are using for analysis:

```
TranlatorPointer<UShort_t> pw(*(rDecoder.getBufferTranslator()), pEvent);
```

After the translation buffer is established we need to read the number of words in the event

```
UShort_t nWords = *pw++;
```

The next couple of lines are used to inform SpecTcl of the size of each event in bytes.

```
CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);
rAna.SetEventSize(nWords*sizeof(UShort_t));
```

We then calculate the number of long words are in the event

```
ULong_t nLongs = (nWords-1)*sizeof(UShort_t)/sizeof(ULong_t);
```

We now have to take care of the bookkeeping details we will need as we unpack an event. We first set the virtual slot number to -1 and define the bool variable FormatOK. Next a 2-dimensional array is defined with indices of channel# and hit#. A second array is defined to keep track of how many hits we have for a given channel, all values in that array are initialized to 0.

```
int nSlot  = -1;
bool FormatOK;
long hits[32][MAXHITSPERCHANNEL];
long nHits[32];
for (int i =0; i < 32; i++) {
  nHits[i] =0;}
```

As long as there are event words left to be read we read the next one and reconstruct the longwords.

```
while(nLongs)
  {
    ULong_t data = *pw;
    ++pw;
    data |= *pw<<16;
    ++pw;
    nLongs--;
```

After a longword has been reconstructed it is tested to determine what kind of word it is. Most types are ignored. If the word is channel data, the channel and the value are decoded. If that channel has less than the maximum hits allowed the value is recorded in the appropriate spot on the 2-dimensional array defined earlier, and the array of hits per channel is incremented. If the longword type is global trailer then we check if it is the last word in the event. If it is then the event is formatted correctly and FormatOK is set to true otherwise it is set to false.

```
      if(CCAENV1x90Data::isGlobalHeader(data)) {
nSlot = CCAENV1x90Data::BoardNumber(data);
      }
      if(CCAENV1x90Data::isTDCHeader(data)) {
      }
      if(CCAENV1x90Data::isTDCTrailer(data)) {
      }
      if(CCAENV1x90Data::isTDCError(data)) {
      }
      if(CCAENV1x90Data::isMeasurement(data)) {
int nChannel = CCAENV1x90Data::ChannelNumber(data, false);
long nValue  = CCAENV1x90Data::ChannelValue(data, false);
if(nSlot == m_nVsn) {
  if(nHits[nChannel] < MAXHITSPERCHANNEL) {
    hits[nChannel][nHits[nChannel]] = nValue;
    nHits[nChannel]++;
  }
}
      }
      if(CCAENV1x90Data::isGlobalTrailer(data)) {
if(nLongs) {
  FormatOK = false;
} else {
  FormatOK = (nSlot == m_nVsn);
}
      }
    }
```

After the event is decoded it is time to assign it a place on the rEvent buffer so that it can be read by SpectTcl to be histogrammed. First it is important to make sure the event Format was OK. If so we check that our trigger signal, going into channel 0, only got one hit. If that is true then the time at which the trigger occurred is recorded as nGateTime. We then cycle through both indices of the 2-dimensional value array. Each value recorded in that array is subtracted from nGateTime and stored in the proper place in the rEvent array. We subtract the nGateTime to compensate for the poor trigger resolution. This is the work around to get good resolution. We then return true indicating that the package was unpacked and we are ready for the next event.

```
  if(FormatOK)
    if(nHits[0] == 1)
      {
      long nGateTime = hits[0][0]; // Gate time.
      int nParam = m_nBaseParameter;
```

```
      for(int nChan = 1; nChan < 2; nChan++) {
        for(int ihit = 0; ihit < nHits[nChan]; ihit++) {
          rEvent[nParam + ihit] = nGateTime - hits[nChan][ihit];
        }
        nParam += MAXHITSPERCHANNEL;
          }
      }
  }
      return kfTRUE;
}
```

3.2.5 Modifying MySpecTclApp.cpp

The next step in preparing SpecTcl is to modify MySpecTcl.cpp to access V1x90Processor to unpack and our module. To do this add the following line to the beginning of the file:

```
#include "V1x90Processor.h"
```

Then find the function

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

and add the line:

```
RegisterEventProcessor(*(new MyEventProcessor(4,100)));
```

Where the first argument is the VME slot Number and the second is the beginning element of the rEvent buffer to look at.

3.2.6 Writing setup.tcl

We can now write the setup script to define the parameters and spectra of interest for SpecTcl. That script should look like:

```
proc Attach {} {
    attach -pipe /usr/opt/daq/Bin/spectcldaq tcp://spdaq22:2602/
    start
}
button .attach -command Attach -text "Attach Online"
pack .attach

set MAXHITS 1

set slot 100
for {set ch 1} {$ch < 16} {incr ch} {
    for {set hit 0} {$hit < $MAXHITS} {incr hit} {
        set name channel$ch
        append name _$hit
        parameter $name $slot 12
        spectrum $name 1 $name {{0 1048575 1048576}}
        incr slot}}
sbind -all
```

3.2.7 Compiling

You are now ready to compile SpecTcl. After fixing any compilation errors start Readout and begin a new run. Then start SpecTcl and run setup.tcl.[1] At this point you should be able to set the Xamine display to see your spectra.

4 Notes about common start

Since the V1290 is only able to accept data for one microsecond after the trigger you will have to delay the trigger in order to use the V1290 as a common start TDC. The way to do this is simple and is illustrated in Figure 5. The trigger delay should be the same as the window offset.

5 More Information

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

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

[1] If while running setup.tcl you get a memory allocation error, you will have to increase the memory allotment for SpecTcl. To do this open SpecTclInit.tcl and increase the value of "set DisplayMegaBytes".

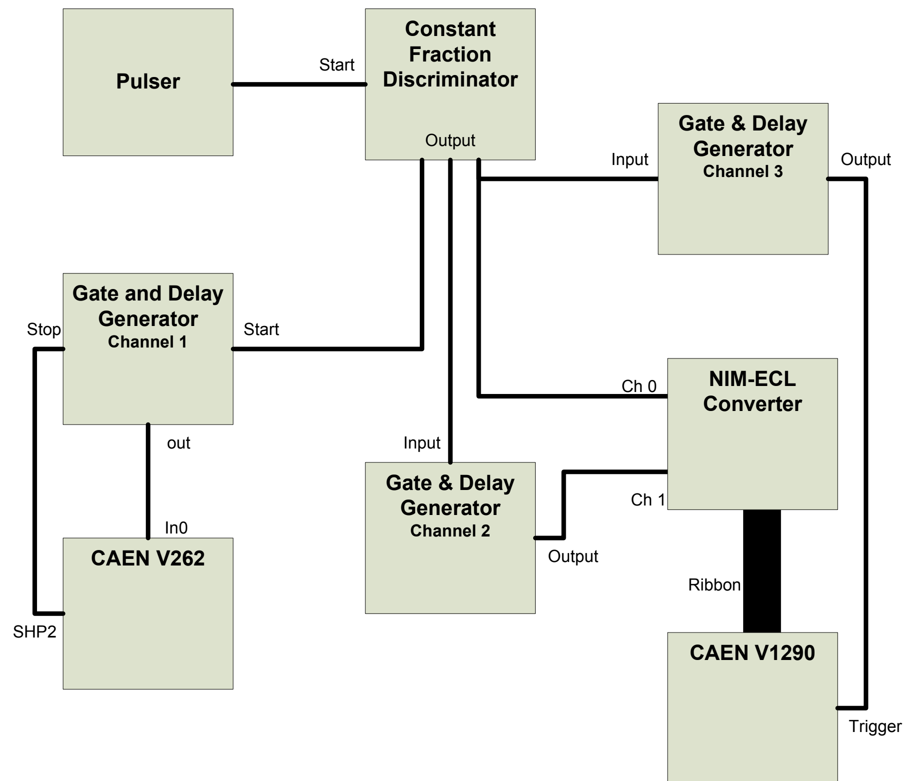

Figure 5: Simple electronics setup for running a V1290 in "Common Start" mode.