Using NSCLDAQ with a CAEN V785 Peak-Sensing ADC and CAEN V262 IO Register

Jeromy Tompkins

Tim Hoagland

Ron Fox

Using NSCLDAQ with a CAEN V785 Peak-Sensing ADC and CAEN V262 IO Register

by Jeromy Tompkins

by Tim Hoagland

by Ron Fox

Revision History

Revision 1.0 11/03/04 Revised by: TH & RF
Original release
Revision 2.0 Feb 5, 2015 Revised by: J.R.T.
Updated for NSCLDAQ 11.0

Table of Contents

1. PREFACE 1

2. The electronics 2

2.1. A minimal electronics setup 2

3. Setting up the software 7

3.1. Modifying the Readout Skeleton 7

3.2. Integrating your event segment with Readout 12

3.3. Compiling the Readout program 12

4. The dumper program 14

4.1. A very brief introduction to ring buffers 14

4.2. Starting up the dumper program 14

5. Running the Readout Program 16

6. Interpreting the dumper output 18

List of Figures

2-1. A simple electronics setup for the CAEN V785 2
2-2. Pulser Output Signal....................3
2-3. Amplifer Output Signal....................4
2-4. Amplified Signal with Logic Pulser....................5
2-5. Amplified Signal and Gate 5

Chapter 1. PREFACE

This paper's purpose is to guide the reader through the process of setting up and reading data from a CAEN V785 peak-sensing ADC using an SBS Bit3 PCI/VME bridge. The trigger for the readout of the device in this tutorial will be initiated by a CAEN V262 IO Register and the data read out will be formatted in a "packet" structure. Furthermore, we will demonstrate how to do this in way that generates data with body headers.

This tutorial will encompass the steps to get the system running from start to finish. In doing so, it will cover how to set up the electronics and how to modify the SBS Readout software skeleton. The targeted version of NSCLDAQ software will be 11.0. Testing for the code is also covered to a limited extent.

This paper assumes that you are

- Somewhat familiar with Linux.
- Familiar with the C++ programming language at a basic level.
- Familiar with an oscilloscope.

The final version of the software is available for download at: *MAKE THE LINK CORRECT!*

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 <`helpme@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.

It will be assumed that the user has a valid NSCLDAQ 11.0 installed at a directory in your PATH. If you have not already done so, do the following before continuing:

```
spdaqxx> unset DAQROOT
spdaqxx> source /usr/opt/nscldaq/11.0/daqsetup.bash
spdaqxx> export PATH=$DAQBIN:$PATH
```

Chapter 2. The electronics

The CAEN V785 is a 12-bit, 32-channel analog-to-digital converter that returns a digitized value of 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/.

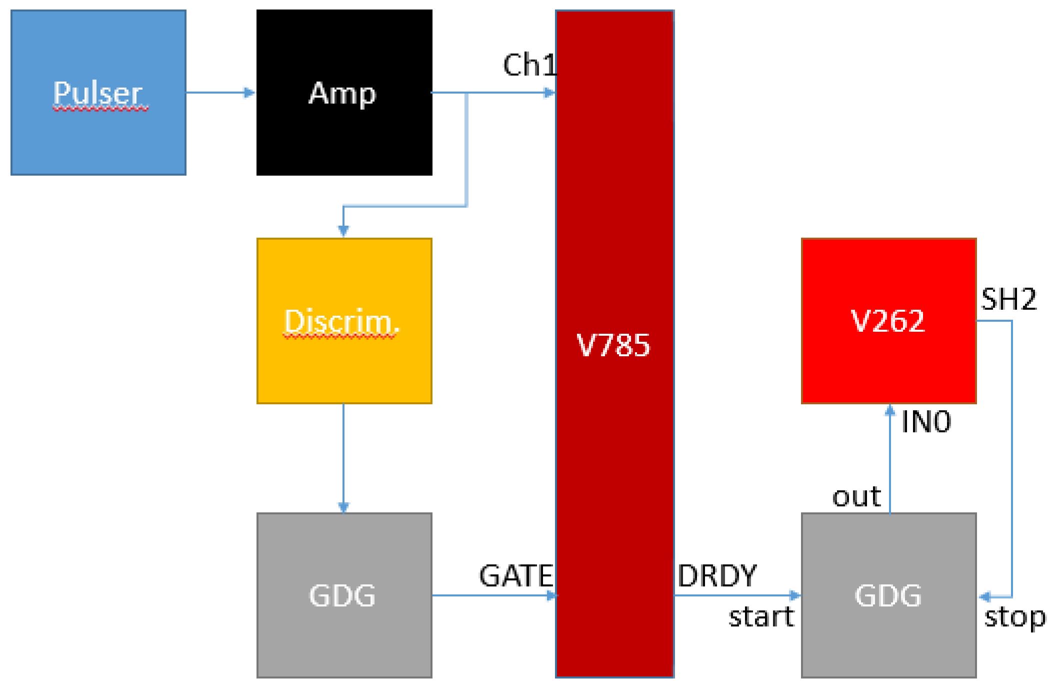
The CAEN V262 is a device that can perform basic functions when an input signal is received. For this setup, a register on the device will be polled by software to determine if a signal is present. You can also find the product manual for this by visiting the manufacturer's website at http://www.caen.it/.

2.1. 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
- Pulser
- 50-Ohm LEMO terminator
- An assortment of LEMO cables
- A 34-conductor ribbon cable with 3M connectors on each end
- An oscilloscope
- A CAEN V262 IO Register

Figure 2-1. 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 pulser 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 2-2. Pulser Output Signal

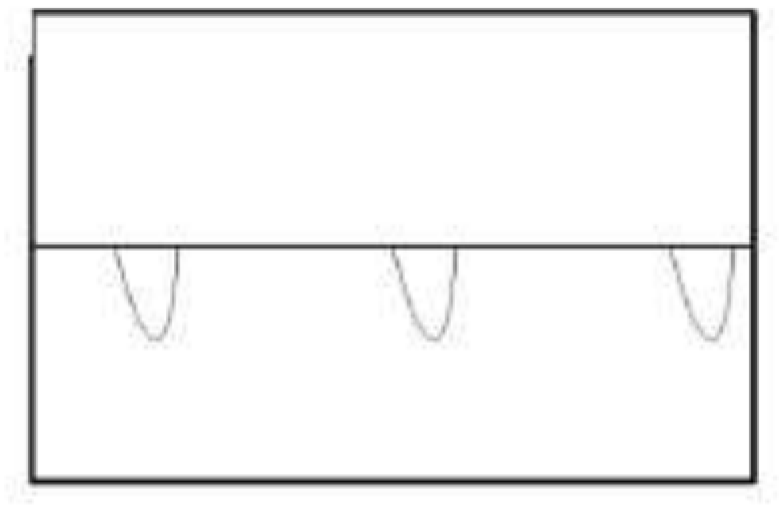
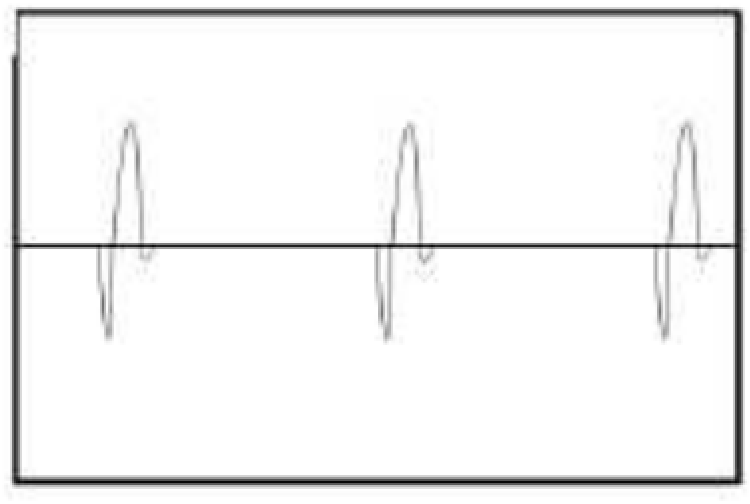


Figure 2-3. Amplifer Output Signal

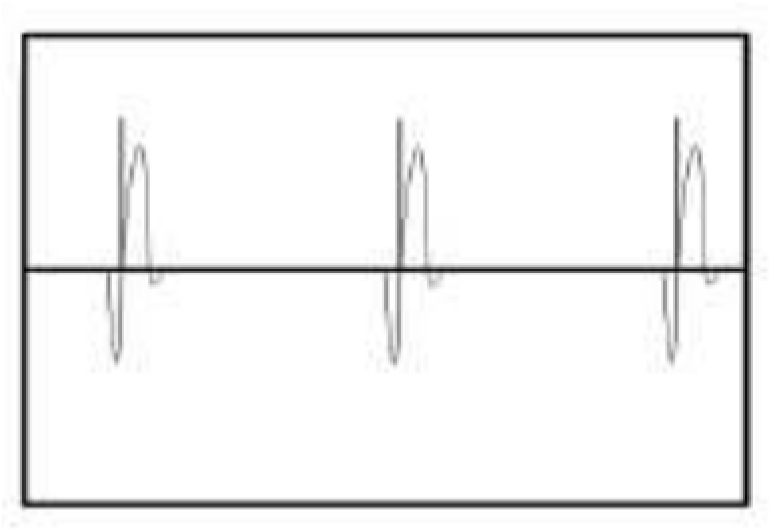


After you are happy with the signal from the amplifier, plug the bipolar output into the splitter. The splitter will simply split the signal much like a "T" but will keep the 50 Ohm termination needed by the pulser.

Take an output of the splitter, convert it to a ribbon cable, and plug that into the channel inputs of the V785. Ribbon cable can be 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 into the correct channel of the ADC. Another useful tip is that the connectors typically have a marking on one end to conventionally identify channel 0.

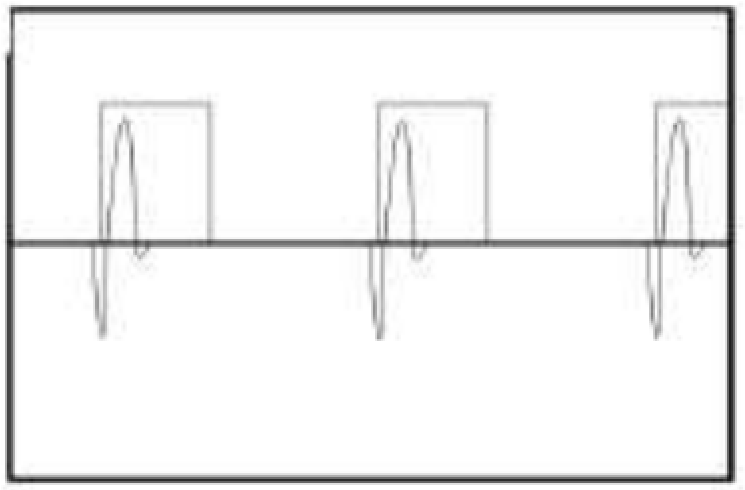
The other output from the splitter will go into a discriminator. This could be either a leading edge or a constant fraction discriminator. The job of the discriminator is to output a logic signal when an input signals has exceeded a certain voltage threshold. The output signal will be used to tell the ADC to initiate the process of recording the peak height of the signal. 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 2-4. Amplified Signal with Logic Pulser



When the threshold is set to a reasonable level, connect one of the discriminator outputs to the start of one channel of your gate and delay generator. 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 2-5. Amplified Signal and Gate



Next we will take the "DRDY" output of the V785 and send that into the start input of a gate delay generator. We will then take the output of that gate delay generator into the IN0 of the CAEN V262. Next you should connect the SHP2 output of the V262 to the stop input of the GDG. This scheme ensures that the signal at INO is held high long enough for the software to recognize it.

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 when we start actually taking data in subsequent sections of this tutorial.

Chapter 3. Setting up the software

The software modifications needed to make the system work can be divided into a few tasks:

1. Defining which pieces of hardware are being used.
2. Defining what to do with the hardware when readout is triggered
3. Defining the conditions for triggering readout

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 $DAQROOT/skeletons/sbs/* .
```

3.1. Modifying the Readout Skeleton

Now that we have obtained a copy of the Skeleton file we can begin 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 that class we will also define a timestamp for our data so that the Readout program will produce a body header. For lack of anything better to in this setup, we will use the event counter as the timestamp. We will not have to write custom software for the trigger condition because a class is already provided by NSCLDAQ that we will make use of.

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 3-1. Header for MyEventSegment

```
#ifndef MYEVENTSEGMENT_H                              ❶
#define MYEVENTSEGMENT_H

#include <CEventSegment.h>
#include <CDocumentedPacket.h>
#include <CAENcard.h>
```

```
/*! \brief A class to read out a V785
 *
 * This class derives from the CEventSegment class and defines the basic
 * functionality we desire of the V785 during read. It will format the data in
 * a documented packet before sending it out.
 *
 */
class MyEventSegment : public CEventSegment               ❷
{
  private:
    CDocumentedPacket m_myPacket;                          ❸
    CAENcard          m_module;                            ❹

  public:
    MyEventSegment(short slot, unsigned short Id);         ❺
    virtual void initialize();                             ❻
    virtual void clear();                                  ❼
    virtual size_t read(void* pBuffer, size_t maxwords);   ❽

  private:
    uint64_t extractEventCount(uint16_t* pEOE);           ❾
};
#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 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.

❸ The m_myPacket is a utility that will automate some of the formatting of the data. It will wrap the data read out of this device in a packet structure. This structure is a 32-bit inclusive size (units=16-bit words) followed by a 16-bit packet id and then a payload of data. 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.

❹ The support software for the CAEN V785 ADC is itself a class: CAENcard. We will create an object of this class in order to access the module.

❺ 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).

❻ 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.

❼ The clear method 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.

❽ The read method is called in response to a trigger. This member must read the part of the event that is managed by this event segment.

❾ We are declaring here a method that will extract the event count from the end of event word added by the V785. This will be used for our timestamp.

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 3-2. Impementation of CMyEventSegment

```
#include <config.h>
#include <string>
#include <stdint.h>

#ifdef HAVE_STD_NAMESPACE                                 ❶
using namespace std;
#endif

// Set the polling limit for a timeout
static unsigned int CAENTIMEOUT = 100;

#include "MyEventSegment.h"                                ❷

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

//constructor set Packet details
MyEventSegment::MyEventSegment(short slot, unsigned short Id):
  m_myPacket(Id,"My Packet","Sample documented packet",pPacketVersion),
  m_module(slot)                                               ❹
{
}

// Is called right after the module is created. All one time Setup
// should be done now.
void MyEventSegment::initialize()
{
  m_module.reset();                                            ❺
  clear();
}

// Is called after reading data buffer
void MyEventSegment::clear()
{
  // Clear data buffer
  m_module.clearData();                                        ❻
}

//Is called to readout data on m_module
size_t MyEventSegment::read(void* pBuffer, size_t maxsize)
{
  // Loop waits for data to become ready
  for(int i=0;i<CAENTIMEOUT; i++) {                         ❼

    // If data is ready stop looping
    if(m_module.dataPresent()) {
      break;
    }
  }

  size_t nShorts = 0;
  // Tests again that data is ready
  if(m_module.dataPresent())
  {
    // Opens a new Packet
    uint16_t* pBufBegin = reinterpret_cast<uint16_t*>(pBuffer); ❽
    uint16_t* pBuf = m_myPacket.Begin(pBufBegin);          ❾

    // Reads data into the Packet
    int nBytesRead = m_module.readEvent(pBuf);             (10)

    // Closes the open Packet
    uint16_t* pBufEnd = m_myPacket.End(pBuf+nBytesRead/sizeof(uint16_t));  (11)

    nShorts = (pBufEnd-pBufBegin);                         (12)

    // set the timestamp
    setTimestamp(extractEventCount(pBufEnd-2));           (13)
  }

  return nShorts;                                          (14)
}

// Extract the lower 24-bits of the end of event word
uint64_t MyEventSegment::extractEventCount(uint16_t* pEOE)
{
    uint64_t count =  *(pEOE)<<16;
    count          |= *(pEOE+1);
    return (count&0x00ffffff);                          (15)
}
```

❶ The lines beginning with the #include of config.h and ending 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 list. Initializer lists specify a list of 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. Lastly, the CAENcard object is constructed using slot number of the V785 for the geographical address of the module.

❺ 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.

❻ This code clears any data that is latched in the module.

❼ The trigger for reading this device out was generated by a separate module that may have been generated prior to the conversion has been completed and thereby ready to read out. For this reason, we will poll the device a finite number of times until the device indicates it is ready to be read out. The CAENcard::dataPresent() function returns true when the module has converted data buffered for read out. At that time, control breaks out of the loop.

❽ The address of the buffer was passed to the method as a pointer without a type. We call this a "void pointer" (i.e. void*). Though useful for passing an address to a generic chunk of memory, there is little else useful that can actually be done with it. To make this more meaningful for filling our buffer, we declare that the pointer is referring to memory segments of 16-bit width (i.e. type = uint16_t). Since the compiler has no way of understanding that this is valid way to treat the buffer, we must flag this as a special type of cast where the memory is reinterpreted to be of a different type. We are basically telling the compiler that we know enough about what we are doing that it should allow us to do so.

❾ 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 32-bits for the word count and then writes the 16-bit packet id. The return value is a "pointer" to the body of the packet.

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

(11) 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.

(12) The number of 16-bit words that have been added to the buffer is computed. It is computed by taking the distance between pointers that reference the original position in the buffer before data was added and the position afterwards.

(13) By calling the CEventSegment::setTimestamp() method, we are ensuring that the event is emitted with a body header. Our solution is to just assign the event count.

(14) Returns the number of 16-bit words added to the data.

(15) The function takes a pointer to the first 16-bits of the end of event word. From the V785 manual, we know that this is a 32-bit word whose lower 24 bits encode the event count. We just need to do some bit-wise arithmetic to extract those 24 bits and return their value.

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

3.2. 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 do 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 <CCAENV262Trigger.h>
#include "MyEventSegment.h"
```

This is necessary because we will be creating an object of class MyEventSegment. We have also included a predefined class for the CCAENV262 IO register. We use this device as a trigger interface.

Next, locate the function `CMyExperiment::SetupReadout()` Modify it to create an instance of `MyEventSegment` and register it to the experiment. In this method we will also instantiate our trigger instance. We provide the base address as the argument to the CCAENV262Trigger constructor which should have been set using the jumper switches on the board to be 0x00100000.

```
void
CMyExperiment::SetupReadout(CExperiment* pExperiment)
{
  assert(pExperiment!=0);
  CReadoutMain::SetupReadout(pExperiment);
  pExperiment->AddEventSegment(new MyEventSegment(10, 0xff00));

  // Register a the trigger module that is situated at base address
  // 0x00100000.
  pExperiment->EstablishTrigger(new CCAENV262Trigger(0x00100000));
}
```

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.

3.3. Compiling the Readout program

Now that the source code for the Readout program is complete, we need to compile it into an executable.

First, 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
```

and 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

Chapter 4. The dumper program

When we run our program, we are going to want to inspect the data that is being read out from the hardware. To do this, we will use the **dumper** program. But first, we need to understand a little bit about a ring buffer (a.k.a. a "ring").

4.1. A very brief introduction to ring buffers

A ring buffer is a fundamental component of the system NSCLDAQ uses to pass data from one process to another and is where the readout program sends its data. Other processes can then read the data from that ring buffer. From the vantage point of the ring buffer, the program filling it with data is its *producer* and the programs that read from it are its *consumers*. There is only allowed to be a single producer per ring buffer while there may be many consumers.

Ring buffers are local to a specific computer but are accessible over the network. Each ring buffer is identified by a user-defined name and the hostname of the computer it is located on. When processes want to attach to a ring to either produce or consume its data, they must specify the name of the ring via a universal resource identifier (URI). The URI specifies the protocol (`proto://`), the hostname (`host`), and the name of the ring (`name`) as a single string: `proto://host/name`. When a user attaches to a ring on another computer, a service running in the background called `RingMaster` sets up the connection that will stream the data across the network for you. In this way, the nework is basically transparent.

4.2. Starting up the dumper program

The program provided by NSCLDAQ for consuming data from a ring buffer and printing it to a terminal is called **dumper**. It is probably the most fundamental diagnostic provided by NSCLDAQ and is incredibly useful in understanding the integrity of a Readout program. We will use it inspect the data outputted by our Readout program.

To start the **dumper** program, log onto any computer on the same network as the one physically connected to the VME crate in which your CAEN V785 has been installed. Typically this is the same machine, but it doesn't have to be. To launch the program, you have to specify the hostname of the computer running the Readout program and also the name of the ringbuffer to connect to. The default values for these are "localhost" and your username. Assuming the Readout program is running locally (i.e. hostname=localhost) and your user name is "user0", you would type either:

```
spdaqxx> dumper --source=tcp://localhost/user0
```

or equivalently,

```
spdaqxx> dumper
```

The dumper terminal session will probably start spewing stuff at to the console once the run starts at a rate that is unreadable. It is generally useful then to limit the number of ring items processed by the dumper by passing the `--count` option. The value provided to this at launch determines the number of items to process before exiting.

You should now attach this to the ring buffer that will contain your Readout program's output. Its name will default to your username. For illustration purposes, we once again assume your username is "user0". Here is how to tell the dumper program to dump the first 10 events that pass through it.

```
spdaqxx> dumper --source=tcp://localhost/user0 --count=10
```

If this failed with some output like this:

```
spdaqxx> dumper --source=tcp://localhost/user0 --count=10
Failed to open data source tcp://localhost/user0
No such file or directory
exiting
```

it just means that the ring buffer has not been created yet. You should create it and then start your dumper program again. Here is how you do that:

```
spdaqxx> ringbuffer create user0
spdaqxx> dumper --source=tcp://localhost/user0 --count=10
```

Chapter 5. Running the Readout Program

The Readout program must run on a system that is physically connected to the VME crate containing your hardware. 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 will show how to use the command line to start your Readout program, start data taking, stop data taking, and exit it.

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 will prompt you for input with a `%` symbol.

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, the initialize and clear member functions of your event segment will be called. The Readout framework will continually check whether your trigger is satisfied and then call the read member function of your event segment when it is.

end

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

Tell the readout program to start a run:

```
% begin
```

After some time passes and you are ready to end the run, you can do so by typing:

```
% end
```

At this point, you can either start a new run in the same way you just did or exit the Readout program entirely. To exit the program, you simply type:

```
% exit
```

Chapter 6. Interpreting the dumper output

When the run started, the **dumper** program should have dumped the data contained in the first ten ring items. Each chunk of data is separated from the next by a row of dashes. The first ring item specified the data format; the second, some information about the transition to actively taking data; the third, documentation about the packet we defined; and the forth, event statistics information. This is effectively what those look like.

```
-----------------------------------------------------------
Ring items formatted for: 11.0
-----------------------------------------------------------
Fri Feb 20 13:56:08 2015 : Run State change : Begin Run at 0 seconds into the run
Body Header:
Timestamp:    18446744073709551615
SourceID:     0
Barrier Type: 1
Title     : Set New Title
Run Number: 0
-----------------------------------------------------------
Fri Feb 20 13:56:08 2015 : Documentation item Packet typesBody Header:
Timestamp:    18446744073709551615
SourceID:     0
Barrier Type: 0
0 seconds in to the run
My Packet:0xff00:Sample documented packet:1.0:Fri Feb 20 13:56:04 2015

-----------------------------------------------------------
No body header
Fri Feb 20 13:56:08 2015 : 0 Triggers accepted as of 0 seconds into the run
 Average accepted trigger rate: -nan events/second
```

The remainder of the output should have been event data that was filled in by the MyEventSegment::read() method. Each of these starts off indicating the size of the event body. They are all 22 bytes long. The next four lines specify the information contained in the body header, and then the body data is the list of 16-bit-wide, hexadecimal numbers that follow. The first two of these words specify the inclusive size of the entire body in units of 16-bit words; it was added by the Readout framework for us. Here is a sample of the first dumped event.

```
-----------------------------------------------------------
Event 22 bytes long
Body Header:
Timestamp:    0
SourceID:     0
Barrier Type: 0
000b 0000 0009 0000 ff00 5200 0100 5001
4e2a 5400 0000
```

You can see that the body header contains correct information. We derived the timestamp from the event count and this is the first event, so it should be zero. You should see that the subsequent events have timestamps that increment by one from one event to the next. The source id is 0, because that is the default value and we didn't specify an alternative. Finally, the barrier type is 0, which is correct for all physics events.

The first 32-bits of the body form the inclusive size. Here that number is 0x0000000b or 11. That makes good sense because we can count 11 16-bit words in the body that was dumped.

The remainder of the body is the packet we defined in our MyEventSegment::read() function. The structure of the documented packet is a 32-bit inclusive size, 16-bit type, and then generic data filled in by the V785. By looking at the manual for the V785, we learn that it adds a series of 32-bit words. So to help understand the data in our packet, I will rewrite it in a clearer way by reordering and grouping these 16-bit chunks. The packet has the form:

Example 6-1. The Packet Data

```
00000009 ❶
ff00     ❷
52000100 ❸
50014e2a ❹
54000000 ❺
```

❶ The 32-bit inclusive size is just 9 16-bit words.

❷ The 16-bit packet type is the number 0xff00 that we defined.

❸ This is the 32-bit buffer header produced by the V785. It encodes the following information: slot=10, pieces of data in buffer = 1.

❹ The only piece of the data in the buffer produced by the V785. This says that the value was 0xe2a or 3626 and that it corresponds to channel 1 of of the module in slot 10.

❺ The final 32-bit word is the end of block marker for the V785. The lower 24-bits form the event count and it specifies it is for the card in slot 10. Because this is the first event, the event count is 0.

The makes complete sense and this we can now rest assured that our system is set up correctly.