# V775 simple setup

**Ron Fox**

**V775 simple setup**

by Ron Fox

Revision History

Revision 1.0 February 19, 2015 Revised by: RF
Original Release

# Table of Contents

**1. Introduction....1**
**2. Setting up the Electronics. ....2**
**3. Creating the Readout program ....3**
3.1. Obtaining the Readout Skeleton....3
3.2. Writing the Event segment....3
3.3. Writing the trigger....7
3.4. Hooking everything together.....9
3.4.1. Addressing VME modules ....10
3.5. Building and testing your Readout....11
**4. Creating the tailored SpecTcl ....14**
4.1. Decoding raw TDC data....14
4.1.1. Getting the skeleton....14
4.1.2. Creating the raw unpacker class ....15
4.1.3. Adding the unpacker to the analysis pipeline....20
4.1.4. Creating raw time spectra....21
4.1.5. Building and testing what we have so far. ....24
4.2. Producing parameters computed from the raw data....25
**5. Running the system....29**
5.1. Readout GUI ....29
5.2. Creating a directory in which to record events ....30
5.3. Running and configuring the ReadoutGUI ....30
5.3.1. A brief tour of the event area structure....31
**6. Taking the example further. ....33**
**7. The full readout program....34**
**8. The full SpecTcl program. ....39**

# List of Figures

2-1. V775 Simple setup electronics block diagram. .....2

# Chapter 1. Introduction

This document will show how to take data from NSCLDAQ, and analyze it with SpecTcl using a CAEN V775 32 channel TDC. We are going to:

- Show a simple electronics setup that will send test pulses into the V775.
- Show how to use the SBS readout framework to read events from the V775
- Show how to use NSCLSpecTcl to create raw spectra for the TDC online.

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.nscl.msu.edu/daq/samples/caenv775/code.zip

# Chapter 2. Setting up the Electronics.

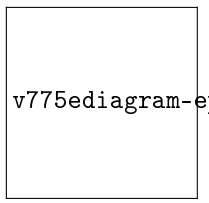
A minimal electronics setup will need to:

- Provide a start signal to the TDC
- Provide a stop signal to the TDC. For this setup we will use a delayed start signal.

In this simple setup we will trigger the computer readout when data are ready in the TDC. Since we only have a single module, its busy output also indicates when the system as a whole is busy.

The electronics diagram below lets us use a simple pocket pulser to run this system:

**Figure 2-1. V775 Simple setup electronics block diagram.**




The discriminator creates a digital pulse from the analog pulser signals. One discriminator output is fed to the `Gate` NIM input of the TDC and will serve as the TDC start signal. A second output is delayed by the Gate and delay generator before being converted to the differential ECL signal expected on the ribbon cable inputs of the V775.

When you connect the ribbon cable between the V775 and NIM->ECL converter, be careful the read the module front panels to ensure that pin 1 on the NIM-> ECL converter goes to pin 1 on the TDC.

# Chapter 3. Creating the Readout program

The readout program interacts with your electronics to read data from your detector system. In response to a *trigger*, it will execute code you write or incorporate to read an event.

We are going to use the SBS readout framework. This framwork requires that the computer you use to read events is attached to your VME crate with an SBS interface card pair connected with an optical fibre cable.

Creating the reaout program requires that you:

- Copy the SBS readout skeleton to an empty directory where you can work on the software
- Create and register an event segment that initializes andreads the V775.
- Create and register a trigger object that uses the V775 data ready status to trigger readout.
- Build and test your tailored readout program.

## 3.1. Obtaining the Readout Skeleton

The readout skeleton is part of NSCLDAQ. For this example we are going to use the most recent version of NSCLDAQ-11.0 We are going to set up environment variables for that version of NSCLDAQ, make a directory called readout in our current working directory and copy the SBS skeleton into that directory:

```
. /usr/opt/daq/11.0/daqsetup                # Define env variables
mkdir readout                               # make an empty directory.
cd readout
cp $DAQROOT/skeletons/sbs/* .               # Copy the skeleton.
```

The Readout skeleton is a template that can be modified to build a functional readout program. In the next two section, we'll do just that.

## 3.2. Writing the Event segment

The Readout program responds to triggers (we'll get to triggers in the next section) by executing one or more event segments to read data from the hardware into an event buffer. You can register as many event segments as you want. Event segments allow you to organize the readout of your experiment into logical chunks. Often event segments will readout one of several detector subsystems that make up your experiment.

Event segments are C++ classes that are derived from the base class `CEventSegment`. The base class defines the interface between your event segment and the SBS readout framework. The base class provides reasonable default implementations for most of these interfaces so you only need to implement the interfaces your event segment needs.

We are going to implement:

- initialize - To set up the V775 for readout.
- read - to read an event.

See `$DAQROOT/include/sbsreadout/CEventSegment.h` and the online reference for `CEventSegment` in http://docs.nscl.msu.edu/daq (http://docs.nscl.msu.edu/daq) for more information about the interface this class defines.

Here is a header that describes the class we are going to write:

**Example 3-1. V775 Event Segment (V775EventSegment.h).**

```
#ifndef _V775EVENTSEGMENT_H                                    ❶
#define _V775EVENTSEGMENT_H

#include <CEventSegment.h>                                     ❷

class CAENcard;                                                ❸



class CV775EventSegment : public CEventSegment                 ❹
{
private:
  CAENcard&   m_module;                                        ❺

public:
  CV775EventSegment(CAENcard& module);
public:
  virtual void initialize();                                   ❻
  virtual size_t read(void* pBuffer, size_t maxwords);
};


#endif
```

❶ Since it is normally a compiler error to define the same entities twice (classes for example) users of your code will thank you for enclosing your headers in a *multiple inclusion protection #ifdef*.

This `#ifdef` and the `#define` that follows it ensures that even if the header is included more than once the code within the body of the `#ifderf` is only seen once by the compiler.

❷ this header is declaring a new class `CV775EventSegmenty` that is derived from the `CEventSegment` base class. To do this properly the compiler needs to know the shape of the `CEventSegment` base class. This class is defined in the `CEventSegment.h` header `#include`d in this line.

❸ The `CAENcard` class provides support for the CAEN V775/V785/V792/V862 digitizer cards. Our class will hold a reference to one of these cards. Since a reference is like a pointer, the compiler does not need to know the full shape of the class at this time so we can make do with declaring the `CAENcard` class as a forward definition.

It's worth asking when to do this as opposed to actually `#include`ing the `CAENcard.h` header. My approach has always been to include only when necessary. This reduces the chances of a circular `#include` dependency in headers and reduces the work the compiler needs to do at any time.

❹ We declare the `CV775EventSegment`. This class is a subclass derived fromt he `CEventSegment` base class. This is important as it lets the Readout framework treat all event segments in a uniform manner letting the system sort out at runtime if a pointer to an `CEventSegment` is really a pointer to a `CV775EventSegment`.

❺ We will construct our event segment by passing it a reference to the `CAENcard` object that is bound to our TDC. Doing this requires that we hold a reference to that `CAENcard` object. That reference is declared as object data by this line.

❻ These lines declare the methods in the base class we will be overriding and implementing in this class.

Let's look at the implementation. We're going to look at the following sections of the implementation file

- File heading `#include` directives etc.
- `Constructor`
- `initialize`
- `read`

**Example 3-2. v775 Event Segment (V775EventSegment.cpp) - file heading**

```
#include "V775EventSegment.h"                    ❶
#include <CAENcard.h>                        ❷
#include <iostream>
#include <stdint.h>


static const unsigned  TDC_RANGE = 0x1e;         ❸
static const unsigned  MAX_WORDS =
        34*sizeof(uint32_t)/sizeof(uint16_t);    ❹
```

❶ Class implementation files need access to the declaration of the class. This `#include` incorporates that declaration.

❷ This module will be invoking methods of the `CAENcard` class so we need to provide the shape of that class to the compiler. This line incorporates the declaration of `CAENcard` into this compilation unit.

❸ The V775 has a variable full scale time. The value chosen, when programmed into the module will result in a 1200nsec full scale value.

❹ This is the maximum number of 16 bit words the TDC can deliver for an event. The module can deliver 32 channels, at a 32 bit word per channel and a header and trailer each 32 bit word long.

**Example 3-3. v775 Event Segment (V775EventSegment.cpp) - Constructor**

```
CV775EventSegment::CV775EventSegment(CAENcard& module) :
  m_module(module)
{}
```

The constructor only has to initialize the reference to the `CAENcard` object that is controlling our TDC.

**Example 3-4. v775 Event Segment (V775EventSegment.cpp) - Initialization**

```
void
CV775EventSegment::initialize()
{
  try {                                          ❶
    m_module.commonStart();                      ❷
    m_module.setRange(TDC_RANGE);                ❸
    m_module.clearData();
  }
  catch (std::string msg) {                      ❹
    std::cerr << "Unable to initialize TDC (V775)\n";
    std::cerr << msg <<std::endl;
    throw;
  }
}
```

❶ Most methods of the `CAENcard` class report errors by throwing exceptions. Placing the entire initialization inside a `try` block allows a coarse grain identification and handling of the error.

❷ The V775 can be either a common start or common stop TDC. This method asks the module to run in common start mode. In that mode the `GATE` input is the start and the individual channel inputs are the stops.

In common stop mode, the individual channels are starts and the `GATE` input is the common stop for all channels.

❸ Sets the time range of the TDC. See the manual for the meaning of this value (section 4.34 `Full Scale Range Register`).

❹ If the initialization failed at any point, control will be transferred to this block. An error message is output to `stderr` and the error is rethrown to be dealt with by our caller. The SBS framework will abort the start of the run if an exception is detected.

**Example 3-5. v775 Event Segment (V775EventSegment.cpp) - reading data**

```
size_t
CV775EventSegment::read(void* pBuffer, size_t maxWords)      ❶
{
  if (MAX_WORDS <= maxWords) {                               ❷
    size_t n = m_module.readEvent(pBuffer);                        ❸
    m_module.clearData();
    return n/sizeof(uint16_t);
  } else {
    throw std::string(
        "CV775EventSegment::read - maxWords won't hold my worst case event" ❹
    );
  }
}
```

❶ The `read` method is called in response to a trigger. The *`pBuffer`* parameter points to storage into which the event segment should store the data it contributes to the event. *`maxWords`* tells `read` how many 16 bit words of data are remaining in that buffer. The method is expected to return the number of 16 bit words of data itactually read.

❷ The device is only read if there is sufficient room in the event buffer. In our simple example, *`maxWords`* will always be much greater than `MAX_WORDS`. This checking is, however good practice as this event segment could appear in conjunction with other event segments that might exhaust the buffer before we are even called.

❸ This block of code reads the digiitzer data into the event buffer and clears it, preparing for the next event. The `readEvent` method returns the number of bytes read, so this must be scaled down to the number of words prior to returning to the caller.

❹ If there is not sufficient room in the buffer, an exception is thrown. Note that the methods in block of code that read the event may also throw an exception.

## 3.3. Writing the trigger

The event segment read methods are called in response to a trigger. But what is a trigger? In the SBS readout framework a trigger is anything we say it is. We supply, or use a trigger object that tells the readout framework when to do the readout.

Usually there is a hardware trigger that is some combination of external hardware and a trigger object that monitors that hardware to determine when the trigger occured. The external trigger hardware then interfaces with the VME crate via either a CAEN V262 I/O register or a CAEN V976 coincidecne register for example.

For our simple setup, the appropriate time to trigger a readout is when the V775 has data available. The `CAENcard` class has a method `dataPresent` that can report this condition. We will use that method as the basis for our trigger class.

In SBS Readout, trigger classes are derived from the `CEventTrigger` base class. As with the `CEventSegment` class, this class provides a set of defined interfaces between the readout framework and objects that can test for trigger conditions.

While there are initialization and tear down methods, our initialization is done by the event segment, so we only need to check for the trigger condition.

Here's the header for our trigger class `MyTrigger`:

**Example 3-6. V775 data ready trigger header (MyTrigger.h)**

```
#ifndef _MYTRIGGER_H
#define _MYTRIGGER_H

#include <CEventTrigger.h>

class CAENcard;

class MyTrigger : public CEventTrigger
{
private:
  CAENcard& m_module;
public:

  virtual bool operator()();
};
```

If you understood how we wrote the event segment this should be clear. The only new wrinkle is the use of `operator()`. This is the function call operator. When defined, it allows objects of a class to be dalled as if they were functions. Objects of this sort, functions that have state are often called *functors*.

The implementation is also simple:

**Example 3-7. V775 data ready trigger implementation (MyTrigger.cpp)**

```
#include "MyTrigger.h"
#include <CAENcard.h>


MyTrigger::MyTrigger(CAENcard& module) :
  m_module(module)
{}

bool
MyTrigger::operator()() {
  return m_module.dataPresent();
}
```

The constructor saves a reference to the module and the function call operator returns the state of that module's data present. If an object of this class is established as a trigger, it will trigger a readout whenever thte module has data.

# 3.4. Hooking everything together.

In the last two sections we've written our event segment and trigger classes. In order to use them

- We must add an instance of our event segment to the Readout framework's top level event segment.
- We must declare an instance of our trigger as the event trigger.

These are done by editing the `Skeleton.cpp` file that you copied into your work directory for Readout.

At the top you will need to `#include` the headers for your new classes:

```
#include <config.h>
#include <Skeleton.h>
#include <CExperiment.h>
#include <TCLInterpreter.h>
#include <CTimedTrigger.h>

#include <CAENcard.h>              // Add this line.
#include "V775EventSegment.h"            // Add this line
#include "MyTrigger.h"                   // Add this line.
```

The `Skeleton.cpp` method `SetupReadout` is where we will make the remainder of the changes. We're going have to:

- Create a long lived `CAENcard` object to communicate with our V775.
- Create and register a `MyTrigger` object to act as the event trigger, that uses the card we created.
- Create and register a `V775EventSegment` to readout the V775 card we created.

Here's what the `SetupReadout` method looks like when we're done modifying it:

```
static const uint32_t V775Base=0x11110000;                    ❶

void
Skeleton::SetupReadout(CExperiment* pExperiment)
{
  CReadoutMain::SetupReadout(pExperiment);

  CAENcard* pModule = new CAENcard(10, 0, false, V775Base);      ❷

  pExperiment->EstablishTrigger(new MyTrigger(*pModule));        ❸

  pExperiment->AddEventSegment(new CV775EventSegment(*pModule)); ❹


}
```

❶ Here we define a constant; `V775Base` to be the base address of the TDC module. See *Addressing VME modules* below for more about addessing VME modules.

❷ The constructor for both the `MyTrigger` and `CV775EventSegment` classes require a reference to a `CAENcard` object. This object must live for the lifetime of the program. Therefore it is dynamically created here. Had we simply declared a `CAENcard module(10,0,false,V775Base)`, that would have been destroyed when the function exitd.

Please see: *Addressing VME modules* below for guidance on how to specify this module. How you parameterize this constructor depends on a few things that I cannot predict.

❸ The `EstablishTrigger` method of the `CExperiment` class is how you register the experiment's trigger. In this case we create and register an instance of our `MyTrigger` trigger class.

The readout framework can only have a single trigger registered. There is, however, nothing to stop you from building a trigger class that incorporates and does some logicn on more than one trigger source to decide if a Readout should be done.

❹ The `AddEventSegment` method of `CExperiment` appends an event segment to the top level event segment. You can add as many event segments as you need.

The top level event segment is a `CCompoundEventSegment` which is simply a container for other event segments that implements the `CEventSegment` interface by interating through its members.

### 3.4.1. Addressing VME modules

The VME bus appears to the computer like chunk of memory. Modules plugged into the VME bus have a *base address* that determines where in this memory space they live. In addition to a base address, the VME bus supports several address spaces through the use of *address modifiers* that are supplied in the address cycle of a transaction.

The CAEN V775 base address can be determined either by a base address set in rotary switches mounted on the module, or, if conditions are right, by its position in the backplane. The latter mechanism is referred to as *geographical addressing*.

Geograhpical addressing is only possible if both the module and the VME backplane have a small, middle connector between the top and bottom VME bus connectors. This connector implements the *VME430* VME bus extension designed at CERN. Geographical addressing requires it because the slot number is encoded in pins on that middle connector, and is used to compute the base address of the module.

When setting up a VME system it is important to ensure that there are no overlaps in the address ranges of the modules qualified by the address spaces in which they live. In our example, we have assumed that we are not using geographical addressing (it's never mandatory) and that the base address rotary switches were set to `0x11110000`.

Each module provides an identifying field in its data. This is also called the `GEO` field. This is because in a VME430 backplane, V775 modules that have the third connector will unconditionally return the slot number for this field. For systems that don't implement VME430, this value is programmable.

Thus the first parameter for the `CAENcard` constructor must be the slot number (for geographical addressing), or some unique number if geographical addressing is not being used (in which case it is programmed into the module's `GEO`) register.

## 3.5. Building and testing your Readout

When you copied the skeleton, you also copied in a starting point for a Makefile for your project. In this section we're going to modify that Makefile so that a build of your Readout program will compile your trigger and event segments and link them into the Readout.

The Makefile defines a symbol `OBJECTS` that is the list of objects that need to be built. It also defines rules for building C and C++ source files to an objet file in a way that allows access to the headers and libraries of NSCLDAQ. In many cases you just need to add the desired objects to the definition of `OBJECTS`:

```
...
#
#  This is a list of the objects that go into making the application
#  Make, in most cases will figure out how to build them:

OBJECTS=Skeleton.o V775EventSegment.o MyTrigger.o
...
```

Once you have made this modification to the `Makefile` just issue the **make** commnad to build your Readout program.

We're going to use the NSCLDAQ dumper pgoram to test our readout program. To do this, you may want to have a copy of the V775 manual handy so that you can make sense of the data that is read.

Assuming the electronics is set up as described in the block diagram and the base address of the V775 has been configured as described above, we should be able to take data from this system

First building and running your Readout interactively:

```
make
./Readout
%
```

In a second terminal window; setup the NSCLDAQ environment definitions as before and:

```
$DAQBIN/dumper
```

Now in the Readout window start a run let it run for a while and end the run.

```
begin
%  (wait a bit)
end
%
```

Let's look at some dumped events for the case where I had a stop input in to channel 0 of the TDC:

```
------------------------------------------------------------
Event 16 bytes long
No body header
0008 0000 5200 0100 5001 41f3 5400 1555

------------------------------------------------------------
Event 16 bytes long
```

```
No body header
0008 0000 5200 0100 5001 41f3 5400 1556

------------------------------------------------------------
Event 16 bytes long
No body header
0008 0000 5200 0100 5001 41f3 5400 1557
```

Before we pick this apart, I want to make a comment about word *endianess*. Endianess, in this context means that when you represent data that are larger than a byte there is a choice (usually made by the hardware) made to determine if the first bytes represent the low order bits (little endian), or the high order bits (big endian).

While the computers we run NSCLDAQ on (intel chips) are little endian, the VME bus is inherently big endian. This will be clear as we describe the meaning of the longwords in the data.

The SBS readout framework will prefix the data you read with a self-inclusive count of the number of 16 bit words in the event. Since this is generated by the SBS readout program running in the intel CPU, this 32 bit item is in little endian order (first the 0x0008 which are the low order bits then 0x0000 the high order bits). According to these data, the event 8 16 bit words in length.

The remainder of the data are as described in the V775 manual. The VME bus is inherently big endian, so the data have the high order bits first:

1. `0x52000100` is a header for virtual slot 10 which has a single channel worth of data.
2. `0x500141f3` is data for channel number 0 with the value `0x1f3` and the valid data bit set.
3. `0x54001557` is a trailer word for event number 0x1557 since the event count was zeroed (a the start of the run).

# Chapter 4. Creating the tailored SpecTcl

Now that the data look right, we're going to tailor SpecTcl to do online analysis of that data. In this example, we are going to write our code as if the module is the only item to decode. We will decode our data in to parameter named `t.0`...`t.31`. In a larger set up you might want to encapsulate the data from the TDC in a *packet* and have your code search for that packet in the data, only unpacking that part of the data.

SpecTcl analyzes data using the model of an analysis pipeline. Each stage of the pipeline has access to the prior stage's data, as well as the raw event. We'll illustrate this by writing a second stage of the pipeline that produces time differences between adjacent channels of the TDC. The interesting thing is that we can write that stage without having any knowledge of the format of the raw event.

This suggests that your SpecTcl software should consists of analysis stages that first decode the raw data and then produce physically useful parameters once those data are decoded. This protects what is usually the hard part of your analysis software from changes to the structure of the raw event.

## 4.1. Decoding raw TDC data

SpecTcl's data analysis pipeline is expected to take a stream of raw events and unpack it into *parameters*. Spectra can then be defined on those parameters. Gates can also be defined on parameters and used to conditionalize when a spectrum is incremented. Each stage of the event analysis pipeline is an object from a class derived from `CEventProcessor`.

In this section we'll write an event processor class that will decode the data our readout program is producing. To do this we will:

- Obtain a copy of the SpecTcl skeleton.
- Write our event processor class.
- Add an object of our event processor to the SpecTcl analysis pipeline.
- Create a SpecTcl startup script that creates raw time spectra.
- Build and test our modified SpecTcl.

### 4.1.1. Getting the skeleton

In this section we are going to create a working directory and copy the SpecTcl-v3.4 skeleton in to that directory.

**Example 4-1. Copying in the SpecTcl-v3.4 skeleton**

```
mkdir spectcl
cd spectcl
cp /usr/opt/spetcl/3.4/Skel/* .
```

## 4.1.2. Creating the raw unpacker class

Here is the header for our raw unpacker (event processor) pipeline stage:

**Example 4-2. Raw TDC unpacker (RawUnpacker.h)**

```
#ifndef _RAWUNPACKER_H
#define _RAWUNPACKER_H
#include <config.h>                                  ❶
#include <EventProcessor.h>                          ❷

class CTreeParameterArray;                                  ❸

class CRawUnpacker : public CEventProcessor                 ❹
{
public:
  CRawUnpacker();                                           ❺
  virtual ~CRawUnpacker();
  virtual Bool_t operator()(const Address_t pEvent,
                            CEvent&         rEvent,   ❻
                            CAnalyzer&     rAnalyzer,
                            CBufferDecoder& rDecoder);
private:
    CTreeParameterArray&  m_times;                          ❼
};


#endif
```

❶ SpecTcl's `config.h` header contains a bunch of definitions used by other headers to make their definitions properly depending on the system on which SpecTcl is installed.

This header must be included prior to any other headers.

❸ The tree paramter package is a useful package that makes defining and accessing parameters simple. We'll use it to define and access the parameters we will create. This lets the compiler know that we will use that class in a way that does not require the compiler to know its shape.

❷ The `EventProcessor.h` header defines the `CEventProcessor` abstract base class. Our class will derive from that base class. This means that the compiler will need to know the shape of that class when compiling our header.

❹ This defines the new class `CRawUnpacker`. This will be our class for unpacking the raw data. Note that it is declared with `CEventProcessor` as a base class.

❺ Our constructor needs to be declared because it cannot be compiler defined. Specifically, we will make a connection to our parameters by allocating and saving a tree parameter array and binding it to our parameters.

❻ The function call operator is invoked for each event. Since our job is going to be to unpack the event, we need to declare this method. This method is pure virtual in the base class and therefore all event processors must declare it.

❼ This reference to a tree parameter array will be used to access our parameters.

Let's fill in the class implementation. We'll do this in sections so that no single sample chunk of code is very large. Note that some sections may be presented out of order for the sake of clarity.

**Example 4-3. v775 raw unpacker implementation includes and defs (RawUnpacker.cpp)**

```
#include "RawUnpacker.h"                                    ❶
#include <TreeParameter.h>                          ❷
#include <TranslatorPointer.h>                      ❸
#include <BufferDecoder.h>
#include <TCLAnalyzer.h>                            ❹
#include <assert.h>

#include <stdint.h>

static const uint32_t TYPE_MASK (0x07000000);
static const uint32_t TYPE_HDR  (0x02000000);
static const uint32_t TYPE_DATA (0x00000000);
static const uint32_t TYPE_TRAIL(0x04000000);

static const unsigned HDR_COUNT_SHIFT(8);                   ❺
static const uint32_t HDR_COUNT_MASK (0x00003f00);
static const unsigned GEO_SHIFT(27);
static const uint32_t GEO_MASK(0xf8000000);

static const unsigned DATA_CHANSHIFT(16);
static const uint32_t DATA_CHANMASK(0x001f0000);
static const uint32_t DATA_CONVMASK(0x00000fff);
```

❶ This include directive includes the header that defines the class we are going to implement. The compiler needs the class definition in order to verify that our implementation methods have the right signatures and refer to defined data in the class.

❷ This header defines the tree parameter package classes. We need it specifically to import the definition of `CTreeParameterArray`.

❸ This class imports definition for the translating pointer. In the discussion of the readout code we made mention of the concept of data endian-ness. Translating pointers are pointer like objets that work with byte order *signatures* within the data and automatically convert data, if needed, to native format from the format of the data in the buffer.

Note that byte order signatures will typically be those of the system writing the event data. If those data, in turn, come from a device with different endian-ness thatn the host system, the user will have to know this and perform additional conversions.

❹ These includes are for miscellaneous headers I am not going to describe in detail.

❺ These constants give symbolic definitions to fields shift counts and values that will be seen in the raw data form the V775. It is best to give symbolic names to values of this sort rather than to just put them into the code where a reader my have trouble identifying their meaning.

**Example 4-4. V775 raw unpacker - getLong utility (RawUnpacker.cpp)**

```
static inline uint32_t getLong(TranslatorPointer<uint16_t>& p)
{
  uint32_t l = *p++ << 16;
  l          |= *p++;

  return l;
}
```

While our redout program runs on a little endian computer, the VME bus is big-endian. This utility takes a translator pointer object that points to a specific 32 bit item in big endian format and converts it to native format.

Note that the translating pointer object is passed by reference. The pointer is incremented to point beyond the uint32_t that was converted.

**Example 4-5. V775 raw unpacker object constructor/destructor (RawUnpacker.cpp)**

```
CRawUnpacker::CRawUnpacker() :                          ❶
  m_times(*(new CTreeParameterArray("t", 4096, 0.0, 4095.0, "channels", 32, 0)))
{}

CRawUnpacker::~CRawUnpacker()
{
  delete &m_times;                                      ❷
}
```

❶ The constructor is required to initialize the reference we have to a tree parameter array. It does this by dynamically creating a new tree parameter array whose base name is `t`. This will create actual parameters named `t.00` ... `t.31`.

❷ If an object is ever destroyed, its tree parameter array musb be destroyed as well to prevent memory leaks.

**Example 4-6. v75 raw unpacker unpacking events (RawUnpacker.cpp)**

```
Bool_t CRawUnpacker::operator()(const Address_t pEvent,
                                CEvent& rEvent,             ❶
                                CAnalyzer& rAnalyzer,
                                CBufferDecoder& rDecoder)
{
  TranslatorPointer<uint16_t> p(*rDecoder.getBufferTranslator(), pEvent); ❷
  CTclAnalyzer& a(dynamic_cast<CTclAnalyzer&>(rAnalyzer)); ❸

  TranslatorPointer<uint32_t>p32 = p;
  uint32_t  size = *p32++;
  p = p32;                                                  ❹
  a.SetEventSize(size*sizeof(uint16_t));

  uint32_t header = getLong(p);
  assert((header & TYPE_MASK) == TYPE_HDR);
  assert(((header & GEO_MASK) >> GEO_SHIFT) == 0xa);     ❺
  int nchans = (header & HDR_COUNT_MASK) >> HDR_COUNT_SHIFT;

  for (int i =0; i < nchans; i++) {                         ❻
    uint32_t datum = getLong(p);
    assert((datum & TYPE_MASK) == TYPE_DATA);
    int channel = (datum & DATA_CHANMASK) >> DATA_CHANSHIFT; ❼
    uint16_t conversion = datum & DATA_CONVMASK;

    m_times[channel] = conversion;                          ❽

  }

  uint32_t trailer = getLong(p);
  assert((trailer & TYPE_MASK) == TYPE_TRAIL);         ❾

  return kfTRUE;
}
```

❶ The `operator()` method of a registered event processor is called for each event. This function has access to the raw event via the *`pEvent`* parameter and access to the unpacked parameters via the *`rEvent`* parameter array, although binding tree parameters and tree parameter arrays to *`rEvent`* is usually simpler.

SpecTcl has two other objects that event processors need. The *rAnalyzer* parameter is a reference to an object that oversees the flow of control through the analysis of the data. It is actually the object that invoked `operator()`. Knowledge of the top level structure of the event data is held in *rDecoder* which, for historic reasons is called a `CBufferDecoder`. It is responsible for picking apart the outer structure of the data and passing type decoded data to the analyzer for dispatch.

The expectation is that the analysis pipeline will:

- Decode the raw event turning it into parameters that are in *rEvent* (tree parameters get automatically bound to elements of *rEvent* before the analysis pipline starts).
- Inform the analyzer about the number of bytes that are in the event.
- Detect and inform the analyzer about failures in the pipeline that should abort its execution and discard the parameters prior to the histogramming pass over the data.

All of the work done by `operator()` is directed at one of these three tasks. Note that in a larger pipeline, the second of these tasks, telling the analyzer the event size, only needs to be performed by one of the pipeline elements. There is also no need for all elements of the analysis pipeline to touch the raw event data and, in complex analysis, usually only a few will.

❷ SpecTcl and NSCLDAQ can run on systems of any endianness. `TranslatorPointer` objects behave somewhat like pointers but automatically translate data from the readout system's bye ordering to the host system's byte ordering. Therefore, to be fully portable, we encourage all access to raw event data to be done via a translating pointer.

This line of code creates a translating pointer for uint16_t data that points to the raw event.

❸ SpecTcl's actual defautl analyzer is a `TcAnalyzer` object. While the analyzer object can be configured by the user normally this is not done.

We need to know the analyzer type because the method used to pass the size of the event back to the analyzer is, unfortunately analyzer dependent. In this line we initialize the variable `a` to be a reference to the anzalyzer. The dynamic cast will throw an exception if the analyzer is not, in fact, a `CTclAnalyzer` or an object from a type derived from `CTclAnalyzer`.

❹ This section of code pulls the first 32 bit item from the event, the event size, and uses the `CTclAnalyzer SetEventSize` to inform the analyzer of the event size.

The code does not care about the byte ordering of the data in the buffer because it creates a `TranslatorPointer<int32_t>` to extract this size. Note that translator pointers of various simple data types can be assigned.

❺ Immediately following the event we should see the header for the V775 data. This code ensures that this is the case. It does this by:

- Using our utility function `getLong` to extract the next 32 bit item from the buffer in host order.
- Ensuring that the type field of that item is that of a header (the assert macros will make the program exit with an error message unless the program is defined with `-DNDEBUG`).

- Ensuring the geographical address field of the item matches the geographical address we programmed into the module.

Once the item is validated as a header, the number of channels of data are extracted from it. Note that in production code, the use of `assert` is probabl not appropriate other alternatives are:

- Throw an `std::string` exception. The analyzer catches those exceptions, and outputs the message to stdout. If the analyzer catches an exception, it aborts the event processing pipeline and does not histogram any parameters tht were extracted at that time.
- Output an error message and return `kfFALSE`. This return value causes the analyzer to abort the event processing pipeline and not to run the histogrammer for the parameters extracted so far.
- In some cases it may even be appropriate to output a messasge and return `kfTRUE`. That stops processing the event data but lets the analyzer continue with the next stage of the pipeline (or with histogramming the parameters unpacked so far if this is the last stage).

❻ This loop unpackes the channel data in the TDC.

❼ The loop first asserts that the items that should contain channel data actually does. It then extracts the TDC channel number an data value from the data.

Note that the data also contains a `Valid` bit. The default programming of the TDC suppresses data for which this bit is not set. If you turn that supression off, you will need to decide what to do with data that are not valid.

❽ Tree parameter array objects mimic arrays to the extent that they support indexing. Therefore setting the actual parameter is as easy as this line of text. In this way, `t.00` is the data from TDC channel 0 and so on.

❾ The data words from the TDC should be followed by a trailer. This code asserts that this is the case.

## 4.1.3. Adding the unpacker to the analysis pipeline

We have code to unpack the TDC. SpecTcl needs to be told to use that code. This is done in the method `CreateAnalysisPipeline` in the skeleton file `MySpecTclApp.cpp`.

That file contains an example event processing pipeline which needs to be deleted. In this section we'll look at the modifications you need to make to `MySpecTclApp.cpp` for our simple setup.

First locate the section of that file that contains `#include` directives. Add the following line after the last `#include`:

```
#include "RawUnpacker.h"
```

That makes our raw event unpacking class `CRawUnpacker` known to the compiler in this file.

Next modify the `CreateAnalysisPipeline` method body to look like this:

```
void
CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
  RegisterEventProcessor(*(new CRawUnpacker), "Raw-TDC");

}
```

`RegisterEventProcessor` adds a new event processor to the end of the analysis pipeline. The first parameter is a reference to the event processor object (an instance of a `CRawUnpacker`). The second parameter is a name to associate with the pipeline element.

The name is an optional parameter, but there is the capability to introspect and to modify the analysis pipeline at run time (adding and removing pipeline elements at specific points in the pipe). The methods that locate a specific pipeline element require a name for that pipeline element.

## 4.1.4. Creating raw time spectra

We have our unpacking code and SpecTcl has an instance of our unpacker as its only analysis pipeline element. If we ran SpecTcl now it could unpack the data just fine but nothing would be done with the unpacked parameters. We also need to define a set of spectra. We are going to write a startup script for SpecTcl that does this and ensure that this script is run by SpecTcl when it starts up.

Before we do this, I want to point out that the Tcl in the name SpecTcl is there because SpecTcl uses an enhanced Tcl interpreter to implement its command language. Tcl is a powerful scripting language with a very simple and regular syntax.

For information about Tcl, and it's graphical user interface languate Tk, see http://www.tcl.tk/doc/. http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html (http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html) is a good online tutorial that can get you up and running with the simple stuff quickly. http://www.tkdocs.com/tutorial/index.html (http://www.tkdocs.com/tutorial/index.html) is a tutorial for Tk if you are interested in building GUIs on top of SpecTcl.

Basing SpecTcl's command language around Tcl and Tk allows you to automate tasks SpecTcl performs as well as tailoring application specific graphical user interfaces (GUIs) on top of the program. Most experimental groups have their own GUIs. In our script we're going to look at two approaches to defining our spectra. One is simple but verbose, the other takes better advantage of Tcl's capabilities and is much more consise but still clear.

**Example 4-7. Defining raw Time spectra the hard way.**

```
spectrum t.00 1 t.00 {{0 4095 4096}}
spectrum t.01 1 t.01 {{0 4095 4096}}
spectrum t.02 1 t.02 {{0 4095 4096}}
spectrum t.03 1 t.03 {{0 4095 4096}}
spectrum t.04 1 t.04 {{0 4095 4096}}
spectrum t.05 1 t.05 {{0 4095 4096}}
spectrum t.06 1 t.06 {{0 4095 4096}}
spectrum t.07 1 t.07 {{0 4095 4096}}
...
spectrum t.31 1 t.31 {{0 4095 4096}}
```

Typing all that was pretty traumatic and error prone wasn't it? Let's first look at the spectrum command, which is used to define, delete and list information about spectra. It is used to create spectra with a general form:

`spectrum` *`name type parameter(s) axis-definition(s)`*

Where

name

Is the name of the spectrum you are creating and must be unique.

type

Is the type of spectrum being created. SpecTcl supports a rich set of spectrum types. Type `1` is a one dimensional specttrum.

parameter(s)

Is a list of parameters that are used to increment the spectrum. The actual meaning of this will vary from spectrum type to spectrum type. A one dimensional spectrum needs only one parameter.

axis-definition(s)

Are a list of three element lists that define the axis ranges and bins on each axis. These spectrum has a single axis definition with a range of `0..4095` and `4096` channels along that axis.

The number of axis definitions depends on the type of the spectrum (e.g. 2d spectra, type `2`), have two axis definitions.

When you were typing this in I hope you were thinking "If Tcl is a scripting language there must be a better way to do this right? (Well actually I'm hoping you didn't bother to type this all in and were waiting for this next version).

Have a look at this:

```
for {set i 0} {$i < 32} {incr i} {
    set name [format t.%02d $i]
    spectrum $name 1 $name {{0 4096 4095}}
}
```

Key things to know when decoding this:

$ substitution

If a `$` precedes a variable name, the value of that variable is substituted at that place in the command prior to executing the command. (e.g. $i < 32).

[] substitution

If a string is enclosed with square brackets, It is considered to be a command and the result of executing that command is substituted right there in the original command prior to execution.

(e.g. [format t.%02d $i]).

format command

The format command is like the C function `sprintf` the first command parameter is a format string that is, essentially a `sprintf` format string. The remaining command parameters are values for the placeholders in this string. The command result is what `sprintf` would have stored in its *`str`* buffer.

For example in **format t.%02d $i** if `i` has the value 3, the format command result would be `t.03`

Now that's something that's much more type-able. Create a file `spectra.tcl` and copy/paste that text into it.

Having created `spectra.tcl` we want to ensure that our SpecTcl will execute the commands in that file when it starts. SpecTcl automatically executes a script named `SpecTclRC.tcl` in the current working directory when it starts running. This script is executed towards the end of initialization, after the analysis pipeline has been created (and in case the tree parameters have been created and bound to actual SpecTcl parameters).

A sample `SpecTclRC.tcl` is provided with the skeleton you copied. Locate the line:

```
splash::progress $splash {Loading SpecTcl Tree Gui} 1
```

Insert the following lines *above* that line:

```
set here [file dirname [info script]]
source [file join $here spectra.tcl]
sbind -all
```

The first line defines the variable `here` to be the directory in which the `SpecTclRC.tcl` file lives. The second line sources the `spectra.tcl` file we created from that directory. The third line binds all spectra into the shared memory region SpecTcl uses to provide its displayer the spectra.

## 4.1.5. Building and testing what we have so far.

Let's see if what we have so far actually works. To do this we need to:

- Modify the skeleton Makefile so that our code will be built and linked to SpecTcl
- Build our tailored SpecTcl
- Run our tailored SpecTcl and attach it to the online data stream.
- Start a run so that we're taking data
- View the spectra SpecTcl creates.

As with the Makefile for the SBS readout program, an `OBJECTS` variable lists the names of the objects we want to build. Edit the Makefile that came with the skeleton you copied and change the definition of `OBJECTS` to look like this:

```
OBJECTS=MySpecTclApp.o RawUnpacker.o
```

To build you tailored SpecTcl you can then type:

```
make
```

Run SpecTcl via the command:

```
./SpecTcl
```

A number of windows will pop up. We're going to use two of them. The window titled `treegui` will be used to connect to the online data. The window titled `Xamine` will be used to look at plots of our spectra.

To attach SpecTcl to the online system; use the `treegui` window and select the Online... menum ite from the Data Source menu at the top of that window. In the dialog that pops up, change the radio buttons at the bottom of the dialot to select `ring11` and click Ok. You are now connected to the online

data coming from the system on which you are logged in (so be sure that system is the one connected physically to your VME crate). You can also acquire data that is taken in a remote host, as long as the system you are logged into is running NSCLDAQ. Simply type the name of that system in the box labeled `Host:` before accepting the dialog.

Now using another terminal window login run the Readout program you had already created, and begin a run. You should see statistics at the bottom of the SpecTcl `treegui` window changing showing that SpecTcl is analyzing data. SpecTcl should not exit (that would most likely show that an assertion failed).

To view a spectrum Click on the Display button at the bottom of the `Xamine` window and select the desired spectrum from the list either by double clicking it or by selecting it and clicking Ok.

If you are using the sample electronics setup, the spectra that have signals should show sharp peaks that correspond to the delay you have set in your gate and delay generator.

## 4.2. Producing parameters computed from the raw data

In this section we are going to write a second event processor. This event processor will be positioned after the raw unpacker we just wrote in the event processing pipeline. It will produce parameters that are the differences of the times different channels of the TDC. To test this event processor you will need to fan out your delayed start so that at least two channels will have data. For more fun, delay the fanned signals so that there is a time difference between those channels.

The purpose of this section is to teach the following concepts:

- How event processors can obtain data from previous stages of the event processing pipeline.
- How to determine if a parameter has been assigned a value by a previous stage of the pipeline

We will produce parameters with names like `tdiff.00.01` which wil be the time difference betweeen channel 0 and 1. For simplicity we will produce parameters like `tdiff.00.00` even though these will always have the value 0. We just won't produce spectra for those parameters.

Let's see what the header for an event processor like this might look like:

**Example 4-8. Header for time difference event processor (Tdiff.h)**

```
#ifndef _TDIF_H
#define _TDIF_H

#include <config.h>
#include <EventProcessor.h>

class CTreeParameterArray;
```

```
class CTdiff : public CEventProcessor
{
public:
  CTdiff();
  virtual ~CTdiff();

 virtual Bool_t operator()(const Address_t pEvent,
                           CEvent&         rEvent,
                           CAnalyzer&      rAnalyzer,
                           CBufferDecoder& rDecoder);
private:
  CTreeParameterArray& m_times;
  CTreeParameterArray* m_diffs[32];

};

#endif
```

All this should look very familiar. The notable difference (Besides the change in the class name) is that in addition to a tree parameter reference for the raw times, we have `m_diffs` is an array of 32 pointesr to `CTreeParameterArray` objects. We use pointers because, without creating a new class for encapsulating an array of `CTreeParameterArray` objects we don't have a good way to initialize an array of references.

The idea of this data structure is that `m_diffs[i]` will be an array of differences between channel `i` and the other channels of the TDC.

We will make our life simple by not considering the problems inherent in allowing copy construction and assignment for a class like this.

Let's look at the implementation of the `CTdiff` class:

**Example 4-9. CTdiff implementation (Tdiff.cpp)**

```
#include "Tdiff.h"
#include <TreeParameter.h>
#include <BufferDecoder.h>
#include <TCLAnalyzer.h>
#include <stdio.h>

CTdiff::CTdiff() :
  m_times(*(new CTreeParameterArray("t", 8192, -4095, 4095, "channels", 32, 0)))
{
  char baseName[100];
  for (int i =0; i < 32; i++) {
    sprintf(baseName, "tdiff.%02d", i);
```

```
    m_diffs[i] =
      new  CTreeParameterArray(baseName, 8192, -4095, 4095, "channels", 32, 0);
  }
}

CTdiff::~CTdiff()
{
  for (int i =0; i <32; i++) {
    delete m_diffs[i];
  }
}


Bool_t CTdiff::operator()(const Address_t pEvent,
                          CEvent& rEvent,
                          CAnalyzer& rAnalyzer,
                          CBufferDecoder& rDecoder)
{
  for (int i = 0; i < 32; i++) {
    if (m_times[i].isValid()) {                          ❶
      for (int j = 0; j < 32; j++) {
        if (m_times[j].isValid()) {                      ❷
          (*m_diffs[i])[j] = m_times[i] - m_times[j]; ❸
        }
      }
    }
  }

  return kfTRUE;
}
```

❶ In order to be able to compute the difference of a pair of parameters we need to know that both parameters have been assigned a value by at least one prior stage of the analysis pipeline. Tree parameters, as well as elements of the *rEvent* vector have a method called `isValid` which returns `true` if this is the case.

This line ensures that the first time has been assigned a value.

❷ This line ensures that the second parameter in the difference has been assigned a value.

❸ If both parameters have been assigned a value, the difference is computed and assigned to the appropriate tree parameter.

Note how all of this is done without needing to know the structure of the raw event data. Should the experiment need to change the hardware in a way that changes that structure this code still works properly. A well structured SpecTcl tailoring should consist of several event processors working together to produce the needed parameters.

Don't forget to add an instance of this class to the analysis pipeline.

We'll leave it as an exercise to create a script that makes spectra and to modify `SpecTclRC.tcl` to source that script into the SpecTcl at startup. The axis specifications of these spectra should be `{{-4095 4095 8192}}` e.g.

# Chapter 5. Running the system.

In this section we are going to run the Readout software with a Readout GUI front end that can manage recording the event data. This mode also supports running the Readout program on a remote system. We will also take you on a brief tour of the way the directories for recorded data are organized.

To prepare for this we will need to:

- Set our account up for password-less login over ssh within the lab.
- Designate a directory tree for event recording and create a symbolic link that points to it.

## 5.1. Readout GUI

The Readout GUI runs your readout program over an ssh pipe. This allows it to start Readout in any system that shares the directory in which your Readout is located. For experimental accounts, this is any system within the building, though clearly you need to run Readout on the system that is attached to the hardware it must access.

This is normally used in production experiments where users are located in the data U's but their Readout will run on spdaq systems in a vault.

In order to run the Readout program over an ssh pipe, you need to first set up your account so that you don't need to provide a password when ssh-ing from linux system to linux system within the lab. This involves creating an authentication key for you as a user, without a passphrase and installing the public key as an authorized key:

**Example 5-1. Setting up ssh for password-less login**

```
ssh-keygen -t rsa

Generating public/private rsa key pair.
Enter file in which to save the key (/home/a/.ssh/id_rsa):Enter
Created directory '/home/a/.ssh'.
Enter passphrase (empty for no passphrase):Enter
Enter same passphrase again: Enter
Your identification has been saved in /home/a/.ssh/id_rsa.
Your public key has been saved in /home/a/.ssh/id_rsa.pub.
The key fingerprint is:
3e:4f:05:79:3a:9f:96:7c:3b:ad:e9:58:37:bc:37:e4
```

In the example above, **Enter** means to use the Enter key on your keyboard. Note that some of the output may be slightly different than what you see.

Next you must add the public key file (`~/.ssh/id_rsa.pub`) to the list of keys that are considered known authentication key:

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

This command appends your public key to the authorized keys list creating it if necessary.

## 5.2. Creating a directory in which to record events

Where you put events depends on the type of account you have and the volume of the data you intend to record:

- If you have an account for a specific experiment, prior to the experiment an event area will be allocated and assigned to your account.
- If you have a test account and have told the helproom that you will need to record and retain a large amount of event data during your tests, your account will be assigned a test event area.
- If you have a test account and don't need to record much data you can record it in a directory in your home diretory tree.

We will assume that you only need to record a small amount of data and that you will create a directory in which to record data data:

**Example 5-2. Creating an event area for a small amount of data**

```
mkdir ~/events
ln –s ~/events ~/stagearea
```

The first command creates the directory in which you will record your data. The second, creates a symbolic link expected by the Readout GUI.

## 5.3. Running and configuring the ReadoutGUI

Now that we have prepared the account we can start the Readout GUI:

```
$DAQBIN/ReadoutShell
```

The Readout GUI can manage several Readout programs that contribute data to an *Event Builder*. The Readout GUI refers to these programs as *Data Sources*. The first time you start the Readout GUI you must specify the set of data sources it manages.

ReadoutGUI stores its state in the event area. If you restart ReadoutGUI with the same event area it will remember the data sources you selected.

To set up your data source, click the Data SourceAdd... menu entry. In the resulting popup dialog select `SSHPipe` from the listbox and click Ok.

Each data source type is parameterized in a manner appropriate to its type. In our case, the parameters we need to fill in in the resulting dialog box are:

`Host name:`

The name of the computer in which Readout will run. If Readout will run in the computer on which ReadoutGUI is running, you can leave the default as `localhost`.

`Readout program:`

Use the Browse... button to locat your Readout program and select it.

Once you have selected the appropriate parameterization for the `SSHPipe` data source type to run your Readut, Click Ok.

Before acquiring data, you must start the data source(s). Click the Start button to do this.

When the Readout program starts successfully, you can click the Begin button to start a run and End button to end the run (Begin becomes the End button when a run is active).

To record a run to disk, check the Record checkbutton before starting the run.

Record a few short runs before continuing to the next section

## 5.3.1. A brief tour of the event area structure

The event area maintains a structure that allows non event data to be associated with event files. The mechanics of doing that are beyond the scope of this (already too long) tutorial. For the sake of this section, it's only important to know that the event area uses symbolic links to provide two views of the data.

The event file view provides a single directory which appears to have all event files (appears because in reality this directory contains only symbolic links to the actual event files).

The directory `~/stagearea/complete` contains the event file view. When analyzing data offline this is usually the directory you will need to see. For experiments that don't store metadata with runs, this is usually the only directory you need to look at

The Run view provides a view of the data in which all of the data associated with each run is in its own directory. The `~stagearea/experiment` directory provides this view.

Within this view you will find a directory for each run that was recorded to disk. If you have not assocated metadata with runs, these directories will only contain the event files for that run, e.g. `~/stagearea/experiment/run1` will contain `run-0001-00.evt`.

If you have associated metadata with the runs, the metadata at the time each run ends is also stored in that run's directory.

# Chapter 6. Taking the example further.

There are several directions in which you can take this example further:

- If you have a set of CAEN 32 bit digitizers (V785, V775, V792, V862) modules you can modify the Readout and SpecTcl to read all of them. In that case you can use an external trigger module such as the CAEN V262 or CAEN V977, or install a cable across the ECL control bus and use the global data ready of one of the modules for a trigger.

  When the ECL control bus is bussed in this way, each module's global data ready is the OR of the data ready of the modules.

- You can setup a busy circuit to block triggers when one or more modules is busy. In the existing, single module setup, the TDC can be used in multi-event mode. With multiple modules, you'll either need to know how to build events from the trigger numbers in the trailers of the individual modules, write a timestamp extractor for the event builder that turns the readout into an event source per module or set up an external busy system that latches on the gate and is reset by the computer when readout is complete (effectively turning off the multi event buffers for the modules).

# Chapter 7. The full readout program.

In this section we will include the full text of the following files:

`Makefile`

The Readout Makefile.

`Skeleton.cpp`

The modified skeleton file.

`V775EventSegment.h`

The header for our event segment class.

`V775EventSegment.cpp`

The implementation of our event segment.

`MyTrigger.h`

The header for our trigger class.

`Mytrigger.cpp`

The implementation of our trigger class.

**Example 7-1. `Makefile`**

```
#
#  This establishes which version of NSCLDAQ we're using and where it's installed:
#  Supposedly  you only need to change this definition to update to a newer
#  version of the software:

INSTDIR=/usr/opt/daq/11.0-rc18

include $(INSTDIR)/etc/SBSRdoMakeIncludes



USERCCFLAGS=
USERCXXFLAGS=$(USERCCFLAGS)

USERLDFLAGS=

OBJECTS=Skeleton.o V775EventSegment.o MyTrigger.o

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

clean:
        rm -f $(OBJECTS) Readout
```

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


help:
        echo make           - Build Readout.
        echo make clean     - Remove products from previous builds.
        echo make depend    - Add header dependencies to Makefile.
```

**Example 7-2. `Skeleton.cpp`**

```
#include <config.h>
#include <Skeleton.h>
#include <CExperiment.h>
#include <TCLInterpreter.h>
#include <CTimedTrigger.h>

#include <CAENcard.h>
#include "V775EventSegment.h"
#include "MyTrigger.h"

CTCLApplication* gpTCLApplication = new Skeleton;

static const uint32_t V775Base=0x11110000;  // Base address of the V775.

void
Skeleton::SetupReadout(CExperiment* pExperiment)
{
  CReadoutMain::SetupReadout(pExperiment);

  CAENcard* pModule = new CAENcard(10, 0, false, V775Base);


  // Establish your trigger here by creating a trigger object
  // and establishing it.

  pExperiment->EstablishTrigger(new MyTrigger(*pModule));


  // Create and add your event segments here, by creating them and invoking CExperiment's
  // AddEventSegment

  pExperiment->AddEventSegment(new CV775EventSegment(*pModule));


}
void
Skeleton::SetupScalers(CExperiment* pExperiment)
{
```

```
  CReadoutMain::SetupScalers(pExperiment);      // Establishes the default scaler trigger.

  // Sample: Set up a timed trigger at 2 second intervals.

  timespec t;
  t.tv_sec  = 2;
  t.tv_nsec = 0;
  CTimedTrigger* pTrigger = new CTimedTrigger(t);
  pExperiment->setScalerTrigger(pTrigger);

  // Create and add your scaler modules here.


}

void
Skeleton::addCommands(CTCLInterpreter* pInterp)
{
  CReadoutMain::addCommands(pInterp); // Add standard commands.
}


void
Skeleton::SetupRunVariables(CTCLInterpreter* pInterp)
{
  // Base class will create the standard commands like begin,end,pause,resume
  // runvar/statevar.

  CReadoutMain::SetupRunVariables(pInterp);

  // Add any run variable definitions below.

}

void
Skeleton::SetupStateVariables(CTCLInterpreter* pInterp)
{
  CReadoutMain::SetupStateVariables(pInterp);

  // Add any state variable definitions below:


}
```

**Example 7-3. `V775EventSegment.h`**

```
#ifndef _V775EVENTSEGMENT_H
#define _V775EVENTSEGMENT_H

#include <CEventSegment.h>
```

```
class CAENcard;



class CV775EventSegment : public CEventSegment
{
private:
  CAENcard&   m_module;

public:
  CV775EventSegment(CAENcard& module);

public:
  virtual void initialize();

  virtual size_t read(void* pBuffer, size_t maxwords);
};
```

**Example 7-4. `V775EventSegment.cpp`**

```
#include "V775EventSegment.h"
#include <CAENcard.h>
#include <iostream>
#include <stdint.h>


static const unsigned TDC_RANGE = 0x1e;         // Corresponds to 1.2usec.
static const unsigned  MAX_WORDS = 34*sizeof(uint32_t)/sizeof(uint16_t);



CV775EventSegment::CV775EventSegment(CAENcard& module) :
  m_module(module)
{}

void
CV775EventSegment::initialize()
{
  try {
    m_module.commonStart();
    m_module.setRange(TDC_RANGE);
    m_module.clearData();
  }
  catch (std::string msg) {
    std::cerr << "Unable to initialize TDC (V775)\n";
    std::cerr << msg << std::endl;
    throw;
  }
}
```

```
size_t
CV775EventSegment::read(void* pBuffer, size_t maxWords)
{
  if (MAX_WORDS <= maxWords) {
    size_t n =  m_module.readEvent(pBuffer);
    m_module.clearData();
    return n/sizeof(uint16_t);
  } else {
    throw std::string("CV775EventSegment::read - maxWords won't hold my worst case event");
  }
}
```

**Example 7-5. MyTrigger.h**

```
#ifndef _MYTRIGGER_H
#define _MYTRIGGER_H

#include <CEventTrigger.h>

class CAENcard;

class MyTrigger : public CEventTrigger
{
private:
  CAENcard& m_module;
public:
  MyTrigger(CAENcard& module);

  virtual bool operator()();
};
```

**Example 7-6. MyTrigger.cpp**

```
#include "MyTrigger.h"
#include <CAENcard.h>


MyTrigger::MyTrigger(CAENcard& module) :
  m_module(module)
{}

bool
MyTrigger::operator()() {
  return m_module.dataPresent();
}
```

# Chapter 8. The full SpecTcl program.

This section gives full listings for the event processor headers and implementations. We also provide a full listing of the `SpecTclRC.tcl` and `spectra.tcl` script we wrote to define raw parameter spectra.

We don't providea full listing of the modified `MySpecTclApp.cpp` file. You can find that in the .zip archive for this setup, however.

**Example 8-1. RawUnpacker.h**

```
#ifndef _RAWUNPACKER_H
#define _RAWUNPACKER_H
#include <config.h>
#include <EventProcessor.h>

class CTreeParameterArray;

class CRawUnpacker : public CEventProcessor
{
public:
  CRawUnpacker();
  virtual ~CRawUnpacker();
  virtual Bool_t operator()(const Address_t pEvent,
                            CEvent&         rEvent,
                            CAnalyzer&      rAnalyzer,
                            CBufferDecoder& rDecoder);
private:
  CTreeParameterArray& m_times;
};


#endif
```

**Example 8-2. RawUnpacker.cpp**

```
#include "RawUnpacker.h"
#include <TreeParameter.h>
#include <TranslatorPointer.h>
#include <BufferDecoder.h>
#include <TCLAnalyzer.h>
#include <assert.h>

#include <stdint.h>

static const uint32_t TYPE_MASK (0x07000000);
static const uint32_t TYPE_HDR  (0x02000000);
static const uint32_t TYPE_DATA (0x00000000);
static const uint32_t TYPE_TRAIL(0x04000000);
```

```
static const unsigned HDR_COUNT_SHIFT(8);
static const uint32_t HDR_COUNT_MASK (0x00003f00);
static const unsigned GEO_SHIFT(27);
static const uint32_t GEO_MASK(0xf8000000);

static const unsigned DATA_CHANSHIFT(16);
static const uint32_t DATA_CHANMASK(0x001f0000);
static const uint32_t DATA_CONVMASK(0x00000fff);


static inline uint32_t getLong(TranslatorPointer<uint16_t>& p)
{
  uint32_t l = *p++ << 16;
  l         |= *p++;

  return l;
}


CRawUnpacker::CRawUnpacker() :
  m_times(*(new CTreeParameterArray("t", 4096, 0.0, 4095.0, "channels", 32, 0)))
{}

CRawUnpacker::~CRawUnpacker()
{
  delete &m_times;
}


Bool_t CRawUnpacker::operator()(const Address_t pEvent,
                                CEvent& rEvent,
                                CAnalyzer& rAnalyzer,
                                CBufferDecoder& rDecoder)
{
  TranslatorPointer<uint16_t>p(*rDecoder.getBufferTranslator(), pEvent);
  CTclAnalyzer& a(dynamic_cast<CTclAnalyzer&>(rAnalyzer));

  TranslatorPointer<uint32_t> p32 = p;
  uint32_t  size = *p32++;
  p = p32;
  a.SetEventSize(size*sizeof(uint16_t));

  uint32_t header = getLong(p);
  assert((header & TYPE_MASK) == TYPE_HDR);
  assert(((header & GEO_MASK) >> GEO_SHIFT) == 0xa);
  int nchans = (header & HDR_COUNT_MASK) >> HDR_COUNT_SHIFT;

  for (int i =0; i < nchans; i++) {
    uint32_t datum = getLong(p);
    assert((datum & TYPE_MASK) == TYPE_DATA);
    int channel = (datum & DATA_CHANMASK) >> DATA_CHANSHIFT;
    uint16_t conversion = datum & DATA_CONVMASK;
```

```
    m_times[channel] = conversion;

  }

  uint32_t trailer = getLong(p);
  assert((trailer & TYPE_MASK) == TYPE_TRAIL);

  return kfTRUE;
}
```

**Example 8-3. spectra.tcl**

```
for {set i 0} {$i < 32} {incr i} {
    set name [format t.%02d $i]
    spectrum $name 1 $name {{0 4095 4096}}
}
```

**Example 8-4. SpecTclRC.tcl**

```
lappend auto_path $SpecTclHome/TclLibs
package require splash
package require img::jpeg



set splash [splash::new -text 1 -imgfile $splashImage -progress 6 -hidemain 0]
splash::progress $splash {Loading button bar} 0

puts -nonewline "Loading SpecTcl gui..."
source $SpecTclHome/Script/gui.tcl
puts  "Done."

splash::progress $splash {Loading state I/O scripts} 1

puts -nonewline "Loading state I/O scripts..."
source $SpecTclHome/Script/fileall.tcl
puts "Done."

splash::progress $splash {Loading formatted listing scripts} 1

puts -nonewline "Loading formatted listing scripts..."
source $SpecTclHome/Script/listall.tcl
puts "Done."

splash::progress $splash {Loading gate copy scripts} 1

puts -nonewline "Loading gate copy script procs..."
source $SpecTclHome/Script/CopyGates.tcl
```

```
puts "Done."

splash::progress $splash {Loading tkcon console} 1

if {$tcl_platform(os) != "Windows NT"} {
        puts -nonewline "Loading TKCon console..."
        source $SpecTclHome/Script/tkcon.tcl
        puts "Done."
}


set here [file dirname [info script]]
source $here/spectra.tcl
sbind -all

splash::progress $splash {Loading SpecTcl Tree Gui} 1

puts -nonewline "Starting treeparamgui..."
source $SpecTclHome/Script/SpecTclGui.tcl
puts " Done"


splash::progress $splash {SpecTcl ready for use} 1

splash::config $splash -delay 2000
```

**Example 8-5. Tdiff.h**

```
#ifndef _TDIF_H
#define _TDIF_H

#include <config.h>
#include <EventProcessor.h>

class CTreeParameterArray;

class CTdiff : public CEventProcessor
{
public:
  CTdiff();
  virtual ~CTdiff();

 virtual Bool_t operator()(const Address_t pEvent,
                           CEvent&         rEvent,
                           CAnalyzer&      rAnalyzer,
                           CBufferDecoder& rDecoder);
private:
 CTreeParameterArray& m_times;
 CTreeParameterArray* m_diffs[32];
};
```

```
#endif
```

**Example 8-6. Tdiff.cpp**

```
#include "Tdiff.h"
#include <TreeParameter.h>
#include <BufferDecoder.h>
#include <TCLAnalyzer.h>
#include <stdio.h>

CTdiff::CTdiff() :
  m_times(*(new CTreeParameterArray("t", 8192, -4095, 4095, "channels", 32, 0)))
{
  char baseName[100];
  for (int i =0; i < 32; i++) {
    sprintf(baseName, "tdiff.%02d", i);
    m_diffs[i] =
      new  CTreeParameterArray(baseName, 8192, -4095, 4095, "channels", 32, 0);
  }
}

CTdiff::~CTdiff()
{
  for (int i =0; i < 32; i++) {
    delete m_diffs[i];
  }
}


Bool_t CTdiff::operator()(const Address_t pEvent,
                          CEvent& rEvent,
                          CAnalyzer& rAnalyzer,
                          CBufferDecoder& rDecoder)
{
  for (int i = 0; i < 32; i++) {
    if (m_times[i].isValid()) {
      for (int j = 0; j < 32; j++) {
        if (m_times[j].isValid()) {
          (*m_diffs[i])[j] = m_times[i] - m_times[j];
        }
      }
    }
  }

  return kfTRUE;
}
```