NSCL Ring buffer DAQ tutorial

Ron Fox

NSCL Ring buffer DAQ tutorial

by Ron Fox

Revision History

Revision 1.0 October 16, 2011   Revised by: RF
Original Release
Revision 1.0 November 16, 2011 Revised by: RF
Add appendix for compatibility mode scripts

Table of Contents

List of Examples

Chapter 1. Introduction

This guide provides tutorial and introduction material for NSCL researchers that needto convert their software from the NSCL Spectrodaq data acquisition system to the Ringbuffer data acquisition system. While the ring buffer data acquisition system (RingDaq) attempts to provide a high degree of source code compatibility with Spectrdaq data acquisition (SDAQ), there are some unavoidable differences.

As with SDAQ, the RingDaq provides a data acquisition framework into which experimenters must add a data source program (called Readout by convention), and an online analysis event unpacker (SpecTcl event processor(s)). This guide shows you how to d create these components from scratch as well as from existing SDAQ components you may already have. Each chapter of this guide is intended to be nearly standalone, with supplemental material provided in the reference guide at http://docs.nscl.msu/edu/daq/ringbuffer (http://docs.nscl.msu/edu/daq/ringbuffer). This allows you to skip to the chapter that has the material you need next to get going:

- First I'll describe how to create a readout program from scratch.
- Following that is material that describes how to convert an existing SDAQ production readout program to a RingDaq readout program.
- Next similar material is provided that describes how to create and adaptor that allows you to use code that you currently have to read data in SDAQ's readout classic framework.
- Finally analyzing data from RingDaq with SpecTcl is described along with the set of change you might have to make to convert existing code from SDAQ to RingDaq.
- In addition to the main material, the first appendix describes the format of data items that are put in the ring by readout programs.
- The second appendix describes how to create user written triggers.
- The third appendix describes a set of utilities that allow you to plug spectrodaq analysis components into the ringdaq system.

Chapter 2. Creating a Readout program from scratch

This section describes how to create a readout program for RingDaq starting from scratch. At present we will describe the process only for the SBS Readout framework, as most people in the NSCL will use this framework in the near future. The SBS readout framework is quite simlar to the SDAQ Production readout framework so you can use much of what you know about that framework when building new software for RingDaq's SBS Readout framework.

In general you will need to follow these steps:

- Obtain the readout skeleton from your RingDaq installation directory tree.
- Create event segments to manage the digitizers attached to your detectors.
- Create scaler banks and scaler module code to read any scaler devices you might have.
- Tie all of this together inthe framework skeleton; instantiating objects you will use and registering them with the framework. At this stage you must also instantiate and register an appropriate trigger module so that the Readout will know when to process an event.
- Modify, as necessary, the `Makefile` that was distributed with the skeleton code and usse it to build an executable.

Throughout this discussion I'm going proceed as if `DAQHOME` is an environment variable that points to the top level directory of the ring buffer DAQ installation. At the NSCL this is typically `/usr/opt/daq/10.0`. If you are not at the institution or if this document is old, the value for `DAQHOME` may differ.

2.1. Obtaining the readout skeleton

The readout skeleton is a starting point for building a tailored Readout program. It is located in `$DAQHOME/skeletons/sbs`. Normally you would start building a new readout program in an empty directory as follows:

Example 2-1. Getting the skeleton

```
mkdir myreadout
cd    myreadout
cp $DAQHOME/skeletons/sbs/* .
```

This sequence of unix shell commands creates a new directory named `myreadout`, makes that the current default directory and copies the readout skeleton into that directory.

The readout skeleton constists of the following files:

`Makefile`
: Makefile that builds the skeleton

`Seleton.cpp`
: Source code for the registration code for the readout framework.

`Skeleton.h`
: Header defining the class implemented by `Skeleton.cpp`

If you examine `Skeleton.cpp` you wont' find a `main` function. This is because the readout framework is an application framework. Application frameworks consist of a main program that is written for you and specific ways to register the presence of application specific code that needs to be called at well defined points in the program's execution.

Using an application framework frees you from having to worry about how your code actually interfaces with the data acquisition system, manager run-state transitions, trigger processing and so on. In the next two chapters we will see how to create code that is application specific and how to register it with the framework so that it is called when we want it to be called.

2.2. Creating an event segment

Event segments are software components that manage the dgitizers associated with a logical part of your experiment. You can create an arbitrary number of event segments and control the order in which they are read. The abstraction of event segments supports the fact that experiments may be composed of re-usable detector subsystems and systems.

At the NSCL for example: One might run an experiment using the SeGA gamma ray spectrometer and the S800 spectrograph. One way to do this is to have a re-usable event segment for the S800, another for SeGA and to register them both with the readout framework to build the experiment.

Event segments are instances of classes (objects) that are derived from the class `CEventSegment`. This section will also describe the `CEventPacket` base class which is an event segment that wraps an event segment into an NSCL tagged packet.

The methods an event segment can implement are:

```
void initialize();
void clear();
void disable();
size_t read(void* pBuffer, size_t maxWords);
const bool isComposite();
```

`initialize`

Is called before data taking starts and is expected to initialize the data taking devices to prepare them and enable them to take data. This method is optional. If omitted the framework does nothing to initialize this event segment.

`clear`

Is called to clear digitizers to prepare them to respond to the next trigger. It is called just prior to waiting for a trigger (at the start of the run after `initialize` is called as well as after each event). This method is optional and if not implemented the framework does nothing for this event segment at clear time.

`disable`

This method is called as data taking is being shutdown. If your devices require any actions to disable them you can perform those actions in this method. One place you might use this would be if you have programmed a user specific trigger based on VME interrupts. You could use the method to disable the interrupts on your trigger device.

This method is optional and the framework will do nothing if it is not implementerd.

`read`

This method is called on each trigger it is expected to read the data from the devices managed by this event segment from the digitizer hardware. Parameters are as follows:

type: void*
parameter: *`pBuffer`*
Purpose: Pointer to storage into which this event segment should store its data. Usually the first thing you will need to do is re-cast this pointer to the appropriate data type.

type: size_t
parameter: *`maxWords`*
Purpose: The maximum number of uint16_t units that can fit in the space pointed to by *`pBuffer`*. Very bad things will happen if you read more than this number of words.

The return value is expected to be the number of uint16_t units of data read by this segment.

Enough theory already. Let’s look at a sample implemetation of an event segment. First the header:

Example 2-2. Simple Event Segment Header

```
#include <stdint.h>    ❶
```

```
#include <CEventSegment.h> ❷
class CAENcard;             ❸


class CCAENEventSegment : public CEventSegment  ❹
{
private:
  CAENcard*              m_pCard;              ❺
public:
  CCAENEventSegment(uint32_t base, uint8_t id, int crate = 0); ❻
  ~CCAENEventSegment();

❼
  virtual void   initialize();
  virtual void   clear();
  virtual size_t read(void* pBuffer, size_t maxwords);
private:
  bool haveEvent();
};
```

❶ The `stdint.h` header contains definitions of standard integer types with known bit widths (e.g. uint16_t).

❷ We need to include this header because our event segment will need to be derived from the `CEventSegment` class.

❸ This is the preferred way to define a class in a header file when the class 'shape' does not need to be known. This *forward class declaration* says that `CAENcard` is a class that will be defined later. Using forward class definitions reduces the chances of building circular dependencies between header files.

❹ As promised the `CCAENEventSegment` derives from the `CEventSegment` class. Only `CEventSegment` derived classes can be registered as event segments with the framework.

❺ This data element is why we needed the forward declaration of `CAENcard`. It is going to be a pointer to a `CAENcard` object we will create in our class constructor. That object will be used to manipulate the CAEN digitizer.

❻ Normally your event segments will want to implement a constructor. In this case we provide the constructor with parameters that specify the module base address, virtual slot number (*`id`*). and VME crate number.

Lets look at the implementation (.cpp) file of the event segment a chunk at a time:

Example 2-3. Event segment front matter

```
#include <config.h>       ❶
#include "CCAENEventSegment.h" ❷

#include <CAENcard.h>     ❸

❹
```

```
#include <string>
#include <stdlib.h>
#include <iostream>
```

❶ All implementation code you write in RingDaq should include `config.h` as the first header. This file provides definitions that other RingDaq headers may need.

❶ Since this file will implement the `CCAENEventSegment` class it needs access to the header so that method prototypes and member data definitions are available to method implementations.

❸ This satisfies the forward reference to the `CAENcard` class we made in the header. Since `CAENEventSegment` is going to call `CAENcard` methods, we'll the compiler will need the actual class definition.

❹ The headers below are standard C/C++ headeres that define funtions and classes we will use in the implementation of this class.

The next code section we will look at contains the constructor and destructor of the event segment. The constructor is invoked when the event segment is created (usually in the `Skeleton.cpp` just prior to registration). The destructor is usually never invoked. However if you have some overarching event segment that, at initialization time, creates other event segments it contains, destructors may be called. In order to allow your code to be embedded in environments you don't initially anticipate, you should write correct constructors for all event segments.

Example 2-4. Event segment constructor and destructor

```
CCAENEventSegment::CCAENEventSegment(uint32_t base, uint8_t id,
                                     int packet, int crate) :
  m_pCard(new CAENcard(id, 0, false, base))  ❶
{

}

CCAENEventSegment::~CCAENEventSegment()
{
  delete m_pCard;   ❷
}
```

❶ Initializes the `m_pCare` member data with a pointer to a `CAENcard` object that will manage the CAEN tdc we are operating with.

At this time, no operations are perfomed on the device itself as we've not yet been asked to initialize it.

❷ If we are ever destroyed we must delete the `CAENcard` object the constructor created or memory and SBS mapping resources will be leaked for each construction/destruction cycle.

Next lets look at the initialization code. In a production environment, this code might open a file and read some configuration data, using that data to figure out how to initialize the device. In keeping with showing the simplest code possible, we are going to hard code all configuration information.

Example 2-5. Event segment `initialize` implementation

```
void
CCAENEventSegment::initialize()
{
  m_pCard->reset();             ❶
  sleep(2);
  for(int i =0; i < 32; i++) {  ❷
    m_pCard->setThreshold(i, 0);
  }
  m_pCard->commonStart();       ❸
  m_pCard->keepOverflowData();
  m_pCard->keepUnderThresholdData();
  m_pCard->setRange(0x1e);


}
```

❶ Prior to doing anything to the TDC it is reaset. after being reset it is necessary to wait a bit for the TDC to become ready for programming. The `sleep` is probably somewhat longer than required. A call to `usleep` for a few milliseconds is probably more appropriate.

❷ Since we are going to accept overflow and underthreshold data, the thresholds are just set to zero. Normally you would process some configuration file at this point to determine actual threshold values to program.

❸ This section of code programs the remainder of the TDC configuration.

The `clear` function is trivial:

Example 2-6. Event segment `clear` implementation

```
CCAENEventSegment::clear()
{
  m_pCard->clearData();
}
```

The heart of the event segment is, of course, the code that reads out the module:

Example 2-7. Event segment `read` method implementation

```
size_t
CCAENEventSegment::read(void* pBuffer, size_t maxwords)
{
  // Maximum number of words is 34*2:

  if (maxwords < 34*2 ) {           ❶
    throw
      std::string(
      "CCAENEventSegment - insufficient buffers space for worst case event");
  }

  for (int i =0; i < 30; i++) {
    if(haveEvent()) break;            ❷
  }
  int n = (m_pCard->readEvent(p))/sizeof(uint16_t); ❸


  return n; ❹
}
bool
CCAENEventSegment::haveEvent()      ❺
{
  return m_pCard->dataPresent();
}
```

❶ When `read` is called it is given a pointer that describes where to put data that has been read; *`pBuffer`* and a size_t *`maxwords`* that describes the amount of space remaining in the buffer in words (uint16_t sized units). This code ensures that the maximum TDC event size will fit in the remaining buffer space, throwing an exception if it won't.

The computation of the largest TDC event size comes from the fact that the TDC has 32 channels, that each event will have a header and trailer, and that each item will be a uint32_t, which uses two uint16_t units of storage.

❷ It is possible the trigger latency will be shorter than the conversion time of the TDC. In this loop we wait for the module to have an event's worth of data. If it never does after 30 tests (each test will be about 2 μsecs), the loop exits anyway.

❸ Reads an event worth of data from the module.

❹ The read function returns the number of bytes of data read from the module. Therefore, `n` is the number of uint16_t words read. That value is returned.

❺ This is a convenience function to determine if the module has an event.

Before leaving the subject of event segments for the scaler readout, I want to touch on two other important classes you can use to help you organize your code:

`CCompoundEventSegment`

The `CCompoundEventSegment` class is an event segment that consists of an ordered list of event segments (including other compound event segments).

This can be used to organize a detector system that consists of many detector elements into a single event segment you can hand to your users

`CEventPacket`

NSCL events are often broken into packets. A packet consists of a header that has a size (note in RingDaq the size is a uint32_t while in SPDAQ, it is a uint16_t), a tag, and following the header, the payload of the packet.

`CEventPacket` allows you to wrap an existing event segment (including a compound event segment), in a packet without the event segment knowing it's being wrapped.

Let's look at a simple example of a compound event segment. Earlier in this section, we've built a class; `CCAENEventSegment` that encapsulated a CAEN 32 channel TDC. Suppose we had an experiment that consisted of several of these TDCs (I know it's a bad example but it prevents me from having to build additional event segments and all I really want to show is the mechanics of using compound event segments.)

We might have code the builds a Compound event segment as follows:

Example 2-8. Using a compound event segment

```
...
    // Create a few CCAENEventSegment objects:
    CCAENEventSegment tdc1(0x10000000, 1);
    CCAENEventSegment tdc2(0x10010000, 2);
    CCAENEventSegment tdc3(0x10020000, 3);

    // Create a compound event segment that reads out tdc1, tdc2, tdc3 in order:

    CCompoundEventSegment tdcs;
    tdcs.AddEventSegment(&tdc1);
    tdcs.AddEventSegment(&tdc2);
    tdcs.AddEventSegment(&tdc3);
...
```

The key to this example is that a `CCompoundEventSegment` has a method named `AddEventSegment` whose parameter is a pointer to an existing event segment. This creates an ordered list of event segments.

Since a `CCompoundEventSegment` is itself an event segment it can be added to another `CCompoundEventSegment` as well. Continuing the example above:

```
...
      // Assume we've made more event segments named adcs and qdcs:

      CCompoundEventSegment myDetector;
      myDetector.AddEventSegment(&tdcs);
      myDetector.AddEventSegment(&adcs);
      myDetector.AddEventSegment(&qdcs);
...
```

The event segment hierarchy you build up can be as deep as you need to capture the needs of your detector system.

Suppose now that `myDetector` in the previous example is a know NSCL detector that has been assigned a packet id of 0x1234. We can now create an event segment that wraps our detector in that packet by using a `CEventPacket` as follows:

Example 2-9. Using `CEventPacket`

```
...
    CEventPacket myDetectorPacket(myDetector, 0x1234,
                                    "My  Detector",
                                    "Packet for the My Detector device"
                                    "V1.0");
```

It's pretty easy to understand what the first two parameters are, a reference to an event packet and the tag. The remaining information are bundled into a documentation event emitted at the start of run that describe the set of event packets that you can expect to see in the run. Each documenation event contains a set of strings, one for each packet you created. Each string is a colon separated set of fields consisting of the short name of the packet (`My Detector` above), The stringified packet id (`0x1234`) as a hexadecimal, a long name of the packet (`Packet for the My Detector device` above), and a packet version which should be changed whenever the format of the packet changes (`V1.0` in the example above), and the date and time the packet object was constructed (e.g. `Tue Oct 18 15:32:20 2011`).

If you want to see what the description string looks like, you can call the `Format` method of the `CEventPacket` object.

2.3. Creating scaler banks and scaler modules

The readout framework supports a hierachical organization of scalers using two classes `CScaler` which is intended to represent a single scaler module (but need not), and `CScalerBank` a `CScaler` into which `CScaler` objects including `CScalerBank` objects can be put.

Thus a `CScalerBank` is to `CScaler` as a `CCompoundEventSegment` is to `CEventSegment` objects.

The interface of a `CScaler` is similar to that of a `CScaler`:

```
  void initialize();
  void> clear();
  void disable();
  std::vector<uint32_t> read();
```

`read`

All other methods are completely analagous to their counterparts in `CEventSegment`. `read` is as well, however it is supposed to return a vector of uint32_t values that are the scalers it reads.

The framework will append this set of values to the final set of scalers it will report to the RingDaq.

So lets look at a simple `CScaler`. In this case we will be building a `CScaler` class that manages a single SIS 3820 scaler module.

First the header for the class:

Example 2-10. `CScaler` header

```

#include <CScaler.h>


class CSIS3820;

class CSIS3820Scaler : public CScaler    ❶
private:
  CSIS3820*   m_pScaler;                 ❷

public:
  CSIS3820Scaler(unsigned long base, int crate = 0); ❸
  ~CSIS3820Scaler();


  // The interface required by the CScaler class:

public:                           ❹
```

```
  virtual void initialize();
  virtual void clear();
  virtual std::vector<uint32_t> read();


};
```

❶ Much of this code should look familiar from the' `CEventSegment` discussion. The `CSIS3820Scaler` class must be derived from the `CScaler` class in order to be registered to read scalers with the readout framework.

❷ This member will be a pointer to a `CSIS3820` object that will be used to talk to the scaler module we are reading.

❸ The constructor declaration provides the ability for us to specify the base address an the VME crate in which the scaler is installed. Note that as for the `CEventSegment` since we are going to dynamically create the scaler module class, we need to declare a destructor to allow us to destroy it when our object is destroyed.

❹ We must declare the functions from the `CScaler` base class that will be implemented by `CSIS3820Scaler`. This must be at least the `read` method as that is abstract in `CScaler`.

As before, let's look at the implementation in logical pieces. The front matterr is a few `#include` directives to pull in the headers we need:

Example 2-11. `CSIS3820Scaler` implementation includes section

```
#include <config.h>
#include "CSIS3820Scaler.h"
#include <CSIS3820.h>
#include <iostream>
#include <vector>
```

The constructor and destructor are pretty simple as well:

Example 2-12. `CSIS3820Scaler` constructor/destructor

```
CSIS3820Scaler::CSIS3820Scaler(unsigned long base, int crate) :
     m_pScaler(new CSIS3820(base, crate))    ❶
{


}


CSIS3820Scaler::~CSIS3820Scaler()
{
  delete m_pScaler;      ❷
}
```

❶ Initializes the `m_pScaler` attribute with a pointer to a dynamically created `CSIS3820` object.

❷ Since the constructor created an object dynamically, the destructor is responsible for `delete`ing it.

Next lets look at the `initialize` and `clear` methods as they are relatively short.

Example 2-13. `CSIS3820Scaler` initialize/clear methods

```
void CSIS3820Scaler::initialize()
{
  m_pScaler->setOperatingMode(CSIS3820::LatchingScaler);  ❶
  m_pScaler->setLatchSource(CSIS3820::LatchVMEOnly);
  m_pScaler->EnableClearOnLatch();                         ❷
  m_pScaler->Enable();                                     ❸
  m_pScaler->Arm();

}

void CSIS3820Scaler::clear()
{
  m_pScaler->ClearChannels();                            ❹
}
```

❶ Sets the scaler to latch mode with the VME being the only source of the latch signal. For latching scalers this is normally the appropriate way to read them. Latch mode allows all scaler values to be "simultaneously" recorded for readout at a single snapshot in time.

❷ This function sets the scaler to clear its external counters when they are latched to the internal module memory. See, however the discussion about clearing below.

❸ Enabling and arming the scalers is what starts them counting input pulses.

❹ This clears all of the external counters of the module.

Clears: The way in which this module is being cleared is not techincally correct. The module supports a clear on latch. As coded, the scalers will be cleared a read time and then a bit later when `clear` is called. This can cause a few counts to be lost over the duration of the run.

The correct way to handl this module is to clear it at initialization time and then not to provide a `clear` function. The code was written this way in order to illustrate the use of a `clear` method rather than to be strictly speaking correct.

The `std::vector` class makes it pretty easy to write the `read` method as well.

Example 2-14. `CSIS3820Scaler` read

```
std::vector<uint32_t>
CSIS3820Scaler::read()
{
  std::vector<uint32_t> result;   ❶
  uint32_t              channels[32];   ❷

  m_pScaler->LatchAndRead(reinterpret_cast<unsigned long*>(channels)); ❸

  result.insert(result.begin(), channels, channels + 32); ❹

  return result;


}
```

❶ The `read` method must return an `std::vector`. This declares storage for that vector.

❷ The `LatchAndRead` function we use below will read the 32 channels of the module into an ordinary storage array. Therefore wwe need to declare one here to hold that output.

❸ Reads the data in to `channels`

❹ The `std::vector::insert` method is then used to put the scaler channel values into the result vector which is returned.

Before leaving this discussion of scalers lets take a short tour of the `CScalerBank` class. This class allows us to group several `CScaler` objects into a single scaler-like object.

Suppose we have several SIS 3820 scaler modules. In a detector system. The example below organizes them into a single scaler bank which we can hand out as our detector system's scaler.

Example 2-15. Creating a scaler bank

```
...
    CSIS3820Scaler sc1(0x80000000);
    CSIS3820Scaler sc2(0x80010000);
    CSIS3820Scaler sc3(0x80020000);

    CScalerBank myDetectorScalers;
    myDetectorScalers.AddScalerModule(&sc1);
    myDetectorScalers.AddScalerModule(&sc2);
    myDetectorScalers.AddScalerModule(&sc3);
...
```

Naturally since, `CScalerBank` objects are themselves derived from `CScaler` they can be added, in turn, to other scaler banks.

2.4. Tying the pieces together

Now that we have an event segment and a scaler module, we must tie this all together into a readout program by registering the appropriate objects with the readout framework. In this section we will:

- Show how to select and specify the event trigger.
- Show how to create and register a documented packet from a few `CCAENEventSegment` objects to respond to the event trigger.
- Show how to create and register several `CSIS3820Scaler` objects, organize them as a scaler bank and register them to be read when the scaler trigger fires. In the process we will also point out how to modify the scaler trigger.

All of these operations involve editing the `Skeleton.cpp` file that was distributed as part of the skeleton.

2.4.1. Specifying the event trigger.

In this section we'll look at how to specify the event trigger for the readout framework. Really we need to specify two things. How to know when an event should be read, and how to indicate to the electronics when the event readout is complete.

Each readout must specify an object that is a `CEventTrigger` as the trigger object. This object is repeatedly asked if an event is ready to be read. Each readout may optionally specify an object that is a `CBusy` object. The readout framework interacts with that object (if specified) to determine how to indicate to the electronics that additional triggers can be responded to.

While you can write your own event trigger and busy classes, the framework comes with support for the CAEN V262 and CAEN V977 as trigger/busy electronics in the form of Busy and trigger classes. See the reference information for more about those. For now, we will set up our readout to trigger and report busy-ness via the CAEN V262 module.

Example 2-16. Sepcifying the trigger/busy

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

#include "CCAENEventSegment.h"


#include "CSIS3820Scaler.h"

#include <CCAENV262Trigger.h>   ❶
#include <CCAENV262Busy.h>
```

```
...
void
Skeleton::SetupReadout(CExperiment* pExperiment)
{
...
  pExperiment->EstablishTrigger(new CCAENV262Trigger(0x444400, 0) ); ❷
  pExperiment->EstablishBusy(new CCAENV262Busy(0x444400, 0));        ❸

...
}
```

❶ This header and the next define the `CV262Trigger` and `CV262Busy` classes which we will be using as trigger and busy classes respectively.

❷ This line of code creates a new `CV262Trigger` object for a module with base address of `0x444400` in VME crate 0. This is the traditional location of this module in the NSCL DAQ. The all to the `EstablishTrigger` method of the `CExperiment` object makes this trigger module the experiment event trigger.

❸ Similarly, this line creates a `CV262Busy` object at the same VME base address and establishes it as the module that will handle and maintain the program's busy state.

2.4.2. Specifying what is read out by an event trigger

The `Skeleton::SetupReadout` method of the skeleton is also where event segment should be registered. You can imagine the `CExperiment` as a `CCompoundEventSegment` in the sense that it implements the `AddEventSegment` method. This method allows you to specify the set of event segments you want to respond to the event trigger.

`CExperiment` and `CCompoundEventSegment` invoke corresopnding methods of the event segments added to them in the order in which they were registered. This allows you to control the exact sequence in which event segments put their data into the output event.

The sample code fragments below:

1. Build a compound event segment from several `CCAENEventSegment` objects
2. Wraps the compound event segment into a `CEventPacket` and adds it to the readout.
3. Wraps a single `CCAENEventSegment` object in a `CEventPacket` and adds that to the readout as well.

Example 2-17. Adding event segments to the experiment

```
...
#include "CCAENEventSegment.h"
#include <CCompoundEventSegment.h>    ❶
#include <CEventPacket.h>
...
void
Skeleton::SetupReadout(CExperiment* pExperiment)
{
...
   CCAENEventSegment* pTdc0 = new CCAENEventSegment(0x10000000, 0);
   CCAENEventSegment* pTdc1 = new CCAENEventSegment(0x11000000, 1);
   CCAENEventSegment* pTdc2 = new CCAENEventSegment(0x12000000, 2); ❷
   CCAENEventSegment* pTdc3 = new CCAENEventSegment(0x13000000, 3);
   CCAENEventSegment* pTdc4 = new CCAENEventSegment(0x14000000, 4);
   CCAENEventSegment* pTdc5 = new CCAENEventSegment(0x15000000, 5);

   CCompountEventSegment* pCompound = new CCompoundEventSegment;   ❸
   pCompound->AddEventSegment(pTdc0);
   pCompound->.AddEventSegment(pTdc1);
   pCompound->AddEventSegment(pTdc2);
   pCompound->AddEventSegment(pTdc3);
   pCompound->AddEventSegment(pTdc4);

   pExperiment->AddEventSegment(new CEventPacket(*pCompound,     ❹
                                                 0xff01, "Compound",
                                                 "Sample compound event segment",
                                                 "V1.0"));
   pExperiment->AddEventSegment(new CEventPacket(*pTdc5,         ❺
                                               0xff02, "Simple",
                                               "Sample simple event segment",
                                               "V1.0"));
..
}
...
```

❶ These headers must be included to get the class definitions we need for our readout definitions. The `CCAENEventSegment.h` header is assumed to hold the class definitions for the `CCAENEventSegment` we developed earlier in the chapter.

❷ Creates the event segments that manage the individual TDC modules. Each module is given a unique virtual slot number and base address.

❸ Creates a compound event segment that contains the first 5 TDC events egments (`pTdc0` through `pTdc4`).

❹ Wraps the compound event segment in a packet whose id will be 0xff01 and whose short name is `Compound`. The event segment is added as the first segment to be read by the readout framework in response to a trigger.

❺ Wraps `pTdc5` in a packet whose id will be `0xff02` and whose short name will be `Simple`. This event packet is added as the second segment to be read in response to a trigger.

To conclude this section, let's look at how the software responds to an event trigger.

When the trigger fires, The Readout framework will first invoke the `read` method of `Compound` event packet. The event packet will save space for the packet size and insert the header. It will then call the `read` method of its event segment, `pCompound`. `pCompound` in turn will invoke the `read` method of each of the event segments it wraps in the order in which they were added: `pTdc0`, `pTdc1`,... `pTdc4`. Once all events egments are read, the compound event segment will compute and fill in the size field of the header.

The readout framework will next invoke the `read` method of the `Simple` event segment. This will perform in the same way as `Compound` except that it will directly call the `read` method of `pTdc`.

Once the event has been read out, the same algorithm will be applied, however the `clear` method will be invoked.

2.4.3. Specifying the scaler readout

Specifying the scaler readout is very similar to specifying the event trigger response. This is done in the method `Skeleton::SetupScalers`.

At present, this method sets up a timed trigger as the scaler readout trigger. The default code sets the scaler readout trigger period to `2` seconds.

You must add code to this method to define the response to the scaler trigger. If you don't want a timed trigger, you can substitite some other trigger object if you have some special application need.

In the example below, several SIS3820 scalers are combined to form a scaler bank. That scaler bank is registered, and then a single SIS3820 module is registered.

Example 2-18. Setting up scaler readout

```
...
#include "CSIS3820Scaler.h>"       ❶
#include <CScalerBank>"
...
void
Skeleton::SetupScalers(CExperiment* pExperiment)
{
  CReadoutMain::SetupScalers(pExperiment);

  ❷
```

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


      ❸

      CSIS3820Scaler* pScaler0 = new CSIS3820Scaler(0x80000000);
      CSIS3820Scaler* pScaler1 = new CSIS3820Scaler(0x80010000);
      CSIS3820Scaler* pScaler2 = new CSIS3820Scaler(0x80020000);
      CSIS3820Scaler* pScaler3 = new CSIS3820Scaler(0x80030000);
      CSIS3820Scaler* pScaler4 = new CSIS3820Scaler(0x80040000);
      CSIS3820Scaler* pScaler5 = new CSIS3820Scaler(0x80050000);

      ❹
      CScalerBank* pBank = new CScalerBank;
      pBank->AddScalerModule(pScaler0);
      pBank->AddScalerModule(pScaler1);
      pBank->AddScalerModule(pScaler2);
      pBank->AddScalerModule(pScaler3);
      pBank->AddScalerModule(pScaler4);


      ❺

      pExperiment->AddScalerModule(pBank);
      pExperiment->AddScalerModule(pScaler5);


  }
  ...
```

❶ Includes the headers we will need in the code that follows. The assumption is that the `CSIS3820Scaler` class header is named `CSIS3820Scaler.h`.

❷ This code sets up a timed trigger for scalers. The line that reads `t.tv_sec = 2;` sets the readout period to `2` seconds. Since the scaler timestamp resolution is whole seconds, you shoule leave the `tv_nsec` field set to zero.

❸ This code creates 6 `CSIS3820Scaler` objects and assigns their addresses to `pScaler0` ... `pScaler5`.

❹ Adds the scalers `pScaler0` through `pScaler4` to the scaler bank. The order in which they are added determines the order in which they will be read.

❺ The scaler bank and remaining scaler module are added to the set of scalers the readout will read/clear when the scaler trigger fires. Once more these are read in the order in which they were registered.

2.5. Editing and using the Makefile

Once you have completed your modifications, and written your code, you need to be able to build it. The `Makefile` distributed with the skeleton is heavily commented. Most of the time you will simply have to tell the Makefile to build your objects and incorporate them into the `Readout` executable it builds.

If we assume that we've written two `.cpp` files and corresponding headers named: `CCAENEventSegment.cpp` and `CSIS3820Scaler.cpp`, We need to modify the `OBJECTS` definition to read as follows:

Example 2-19. Makefile modifications

```
                   ...
OBJECTS=Skeleton.o CCAENEventSegment.o CSIS3820Scaler.o
                   ...
```

Chapter 3. Creating a Readout program from a spectrodaq production readout program

This chapter describes how to create a RingDaq readout program given a spectrodaq production readout program as a starting point. The RingDaq readout program closely mimics the SPDAQ production readout framework. Therefore the concepts should seem quite familiar.

In this chapter we'll see how to:

- Obtain a copy of the RingDaq readout framework skeleton
- Port an existing event segment to RingDaq
- Modify the `Skeleton.cpp` to register our event segments.
- Port existing scaler objects to RingDaq and instantiate/register them with `skeleton.cpp`
- Modify the `Makefile` to add our software to the final `Readout` program produced by it.

3.1. Obtaining a copy of the RingDaq readout skeleton

In this section we are going to operate as if an environment variable named `DAQROOT` is defined and points to the top level of the RingDaq distribution. At the time this is being written, at the NSCL this would give `DAQROOT` the value `/usr/opt/daq/10.0`. As time goes on, this directory name may change as version numbers change. If you are not at the NSCL you will need to contact your system administrators about where they installed this software.

The commands below show how to obtain a copy of the readout skeleton for RingDaq:

Example 3-1. Getting the skeleton

```
mkdir myreadout
cd    myreadout
cp $DAQROOT/skeletons/sbs/* .
```

This sequence of unix shell commands creates a new directory named `myreadout`, makes that the current default directory and copies the readout skeleton into that directory.

The readout skeleton constists of the following files:

`Makefile`

Makefile that builds the skeleton

`Seleton.cpp`

Source code for the registration code for the readout framework.

`Skeleton.h`

Header defining the class implemented by `Skeleton.cpp`

If you examine `Skeleton.cpp` you wont' find a `main` function. This is because the readout framework is an application framework. Application frameworks consist of a main program that is written for you and specific ways to register the presence of application specific code that needs to be called at well defined points in the program's execution.

Using an application framework frees you from having to worry about how your code actually interfaces with the data acquisition system, manager run-state transitions, trigger processing and so on. In the next two chapters we will see how to create code that is application specific and how to register it with the framework so that it is called when we want it to be called.

3.2. Porting existing event segments to RingDaq

In this section we will look at how to take an existing event segment for the SPDAQ production readout framework and port it to RingDaq. Let's start by comparing the interfaces of the two types of event segments

The `CEventSegment` class in the SPDAQ system looks like this:

```
void Initialize();
DAQWordBufferPtr& Read(DAQWordBufferPtr& rBuffer);
void Clear();
unsigned int MaxSize();
```

`Initialize`

Provides initialization code that is called at the start of data taking.

`Read`

Reads data in response to the trigger. *`rBuffer`* is a reference to the spectrodaq buffer pointer object that describes where to put the data to be read. The return value is a spectrodaq buffer pointer objedt that has been advanced so that it points to the first word (uint16_t) following the data read by this event segment

`Clear`

Called prior to being able to accept a trigger (including the first trigger). This method is supposed to do any cleanup to make the digitizers able to accept a new event.

`MaxSzie`

This is supposed to returnt he maximum number of uint16_t data alements the event segment will read.

By comparison, the RingDaq `CEventSegment`, which has the same purpose has the following interface:

```
void initialize();
void clear();
void disable();
size_t read(void* pBuffer, size_t maxWords);
const bool isComposite();
```

`initialize`

Is called before data taking starts and is expected to initialize the data taking devices to prepare them and enable them to take data. This method is optional. If omitted the framework does nothing to initialize this event segment.

`clear`

Is called to clear digitizers to prepare them to respond to the next trigger. It is called just prior to waiting for a trigger (at the start of the run after `initialize` is called as well as after each event). This method is optional and if not implemented the framework does nothing for this event segment at clear time.

`disable`

This method is called as data taking is being shutdown. If your devices require any actions to disable them you can perform those actions in this method. One place you might use this would be if you have programmed a user specific trigger based on VME interrupts. You could use the method to disable the interrupts on your trigger device.

This method is optional and the framework will do nothing if it is not implementerd.

`read`

This method is called on each trigger it is expected to read the data from the devices managed by this event segment from the digitizer hardware. Parameters are as follows:

type: void*
parameter: *`pBuffer`*
Purpose: Pointer to storage into which this event segment should store its data. Usually the first thing you will need to do is re-cast this pointer to the appropriate data type.

type: size_t
parameter: *`maxWords`*
Purpose: The maximum number of uint16_t units that can fit in the space pointed to by *`pBuffer`*. Very bad things will happen if you read more than this number of words.

The return value is expected to be the number of uint16_t units of data read by this segment.

The following are a few general remarks about how to port from SPDAQ to RingDaq for each method in `CEventSegment`. It is important to note that the header `spectrodaq.h` does not exist in RingDaq and `#include` directives for it should be removed:

`Initialize`

Change the name of this function to `initialize`. Typically no other changes will be needed.

`Clear`

Change the name of this method to `clear`. Typically no other changes are needd.

`MaxSize`

This method has no counterpart in the RingDaq system. Remove it from your event segment.

`Read`

This method needs the most work:

1. Rename the method to `read` changing the parameter signature to match that of the RingDaq event segment (accepting a void* and a size_t).
2. Usually you will need to cast the input pointer to a uint16_t*. Then replace all arithmetic involving `DAQWordBufferPtr::GetIndex()` with direct pointer arithmetic
3. Return the number of words read rather than a pointer to the next location.
4. Peform a test at the top of the function to see if your worst case event (or if you can determine it your atual event size) is less than or equal to the *`maxWords`* parameter and throw an exception if not.

Let's see how this works in practice. The following two examples show a header and an implementation of an SPDAQ event segment that manages a CAEN V775 TDC.

Example 3-2. SPDAQ Production readout event segment header

```
#include <CEventSegment.h>
#include <stdint.h>

using namespace std;
#include <spectrodaq.h>


class CAENcard;

class MyEventSegment :  public CEventSegment
{
 private:
  CAENcard* m_pCard;
 public:
  MyEventSegment(uint32_t base,  uint8_t id, int crate= 0);
  virtual ~MyEventSegment();

  virtual void Initialize();
  virtual DAQWordBufferPtr& Read(DAQWordBufferPtr& rBuffer);
  virtual void Clear();
  virtual unsigned int MaxSize();

};
```

Example 3-3. SPDAQ production readout event segment implementation

```
#include <config.h>
#include "MyEventSegment.h"
#include <CAENcard.h>


MyEventSegment::MyEventSegment(uint32_t base, uint8_t id, int crate) :
  m_pCard(new CAENcard(id, 0, false, base))
{}

MyEventSegment::~MyEventSegment()
{
  delete m_pCard;
}

void MyEventSegment::Initialize()
{
  m_pCard->reset();
  sleep(2);
  for(int i =0; i < 32; i++) {
    m_pCard->setThreshold(i, 0);
  }
  m_pCard->commonStart();
  m_pCard->keepOverflowData();
  m_pCard->keepUnderThresholdData();
  m_pCard->setRange(0x1e);

  m_pCard->clearData();
}

void MyEventSegment::Clear()
{
  m_pCard->clearData();

}

DAQWordBufferPtr&
```

```
MyEventSegment::Read(DAQWordBufferPtr& rBuffer)
{
  for (int i =0; i < 30; i++) {
    if(m_pCard->dataPresent()) break;
  }
  rBuffer +=(m_pCard->readEvent(rBuffer))/sizeof(int16_t);

  return rBuffer;
}

unsigned int
MyEventSegment::MaxSize()
{
  return 34*2;
}
```

After following the previous suggestions the resulting header for the event segment looks like this:

Example 3-4. Porting the event segment to RingDaq

```
#include <CEventSegment.h>
#include <stdint.h>


❶

using namespace std;


class CAENcard;

class MyEventSegment :  public CEventSegment
{
 private:
  CAENcard* m_pCard;
 public:
  MyEventSegment(uint32_t base,  uint8_t id, int crate= 0);
  virtual ~MyEventSegment();

  virtual void initialize();  ❷
  virtual size_t read(void* pBuffer, size_t maxWords); ❸
  virtual void clear();      ❹
❺

};
```

❶ This note is for what is not there. The `#include` for `spectrodaq.h` has been removed as there is no corresponding header in RingDaq.

❷ In RingDaq the name of this method is entirely lower case rather than starting with an upper case letter.

❸ Note the change not only in name but in parameter types and number.

❹ The clear function has been renamed to be fully lower case.

❺ The `MaxWords` method is not used by RingDaq and has therefore been removed.

Let's take the implementation file in two pieces. First we'll look at all methods other than the `read` method. Then we'll look at the `read` method by itself.

Example 3-5. Porting the Event segment to RingDaq II

```
#include <config.h>
#include "MyEventSegment.h"      ❶
#include <CAENcard.h>
#include <string>


MyEventSegment::MyEventSegment(uint32_t base, uint8_t id, int crate) :
  m_pCard(new CAENcard(id, 0, false, base))  ❷
{}

MyEventSegment::~MyEventSegment()    ❸
{
  delete m_pCard;
}

void MyEventSegment::initialize()    ❹
{
  m_pCard->reset();
  sleep(2);
  for(int i =0; i < 32; i++) {
    m_pCard->setThreshold(i, 0);
  }
  m_pCard->commonStart();
  m_pCard->keepOverflowData();
  m_pCard->keepUnderThresholdData();
  m_pCard->setRange(0x1e);

  m_pCard->clearData();
}

void MyEventSegment::clear()    ❺
{
  m_pCard->clearData();

}
...
```

As you can see not very many modifications were required for this part of the code:

❶ The <string> header was included so that the `read` method (see below) can throw a `std::string` exception.

❷ No changes required to the consturctor.

❸ No changes were required to the destrutor.

❹ The only change required here was to change the first letter of the function name to lower case.

❺ the only change required to this function was to change the first letter of the function name to lower case.

The bulk of the changes are due to the change in the parameter signature of the `read` method:

Example 3-6. Porting the Event segment to RingDaq II

```
...
size_t
MyEventSegment::read(void* pBuffer, size_t maxWords) ❶
{

  if (34*2 > maxWords) {        ❷
    throw std::string("Insuficient space remaining in buffer");
  }

  for (int i =0; i < 30; i++) {
    if(m_pCard->dataPresent()) break;  ❸
  }
  size_t n = (m_pCard->readEvent(pBuffer))/sizeof(int16_t); ❹

  return n;  ❺
}
❻
```

❶ This line had to be changed. The first character of the name of the method was changed to lower case and, instead of a `DAQWordBufferPtr&` return value, a size_t is returned indicating the number of words read. Furthermore, instead of a `DAQWordBufferPtr&` parameter, the function now is passed a void* and a size_t.

❷ As recommended earlier, if the largest amount of data we will produce is larger than our remaining event storage space (*`maxWords`*), a `std::string` exception is thrown.

❸ This code is unchanged.

❹ This code is only slightly changed. Instead of computing the next buffer location, we just need to know how much data were read.

❺ Return the numger of words read.

A few points need to be covered prior to leaving this section:

- If your event segment is more complicated it is often necessary to cast the void* pointer to something else. Suppose, for example in the code we've been working on we had two adc modules m_pCard1 and m_pCard2. We need to know where to put the data for card 2. This can be accomplished as follows:

  ```
  uint8_t* pByteBuffer = reinterpret_cast<uint8_t*>(pBuffer);
  pByteBuffer += m_pCard1->readEvent(pByteBuffer);
  pByteBuffer += m_pCard2->readEvent(pByteBuffer);
  ```

- Sometimes it can be easier to use pointer arithmetic to figure out the number of words read. In the example in the previous point, we could do this as follows:

  ```
  return reinterpret_cast<uint16_t*>(pByteBuffer)
       - reinterpret_cast<uint16_t*>(pBuffer);
  ```

  Subtracting the two uint16_t* pointers gives the number of words between them.

3.3. Registering event segments with `Skeleton.cpp`

You must do three things in `Skeleton.cpp`. You need to select an event trigger, a dead time management scheme (busy) and you need to register your event segment(s).

In the production readout program, the event trigger was built in and you had to do something special to replace it with one that was not built in. As people developed other ways to trigger their readouts this became cumbersome. Therefore the RingDaq readout software requires you to explicitly select the trigger you want to use.

The RingDaq triggers supports two trigger modules directly: `CCAENV262Trigger` and `CV977Trigger` which support triggers from the CAEN V262 and CAEN V977 input registers respectively. In addition, the device support software provides a class (`CCAMACTrigger`) that can easily be wrapped into an event trigger that supports the `IT2` input of the CES CBD8210 CAMAC branch highway driver as a trigger. The base class `CEventTrigger` supports the creation of custom event triggers.

The RingDaq readout framework also provides `CCAENV262Busy` and `CCAENV977Busy` classes that allow those two modules to do dead-time management. The base class `CBusy` supports the creation of custom dead-time management schemes.

In the example we are going to give, we will:

- Set up the CAEV V262 module to handle triggers and dead-time management.
- Create and register an instance of the `MyEventSegment` we ported in the example in the previous section of this tutorial.

Example 3-7. Setting up triggers and registering an event segment

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

#include <MyEventSegment.h>      ❶
#include <CCAENV262Trigger.h>
#include <CCAENV262Busy.h>

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

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

  pExperiment->EstablishTrigger(new CCAENV262Trigger(0x444400)); ❷
  pExperiment->EstablishBusy(new CCAENV262Busy(0x444400));

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

  pExperiment->AddEventSegment(new MyEventSegment(0x10000000, 0xa5)); ❸
}
```

❶ The three headers below are required for our modifications to the skeleton code. We need `MyEventSegment.h` in order to register our event segment and the two `CCAENV262....` headers to specify the trigger and busy devices.

❷ These lines are our first additions to the `SetupReadout` method. They create and register the appropriate objects to use a CAEN V262 at base address `0x444400` in VME crate 0 the trigger and busy management module.

❸ Creates and registers our event segment to respond to the event trigger.

To users of the production readout, this should be familiar territory, with the exception of the need to explicitly register trigger and busy management objects.

3.4. Porting scaler readout to RingDaq

Scaler readout classes area also supported by RingDaq. This section describes how to port an SPDAQ Production readout scaler class to RingDaq and how to register it with the readout skeleton.

Let's start out with a scaler class that can read out a SIS 3820 scaler module for the SPDAQ production readout framework. The header for the starting point of our work is shown below:

Example 3-8. Production readout scaler class (header)

```
#include <CScaler.h>
#include <stdint.h>


class CSIS3820;

class MyScaler : public CScaler
{
private:
  CSIS3820* m_pScaler;
public:
  MyScaler(uint32_t base, unsigned vmeCrate=0);
  virtual ~MyScaler();

  void Initialize();
  void Read(std::vector<unsigned long>& scalers);
  void Clear();
  unsigned int size();

};
```

The implementation of this event segment is shown below.

Example 3-9. Production readout scaler class (implementation)

```
#include <config.h>
#include "MyScaler.h"
#include <CSIS3820.h>

MyScaler:: MyScaler(uint32_t base, unsigned vmeCrate) :
  m_pScaler(new CSIS3820(base, vmeCrate))
{}

MyScaler:: ~MyScaler()
{
  delete m_pScaler;
}
```

```
void
MyScaler::Initialize()
{
  m_pScaler->setOperatingMode(CSIS3820::LatchingScaler);
  m_pScaler->setLatchSource(CSIS3820::LatchVMEOnly);
  m_pScaler->EnableClearOnLatch();
  m_pScaler->>nable();
  m_pScaler->Arm();
}

void
MyScaler::Clear()
{
  m_pScaler->ClearChannels();
}

unsigned int
MyScaler::size() { return 32; }

void
MyScaler:: Read(std::vector<unsigned long>& scalers)
{
  uint32_t              channels[32];

  m_pScaler->LatchAndRead(reinterpret_cast<unsigned long*>(channels));
  scalers.insert(scalers.end(), channels, channels + 32);
}
```

As we will see scaler classes in RingDaq are quite similar:

1. Function names are the same but begin in lower case rather than upper-case (e.g. `Initialize` should be changed to `initialize`).
2. The `read` method simply returns an `std::vector<uint32_t>` rather than appendint to one passed in.
3. There is no `size` method

A straightforward applictation of these rules leads to a header that looks like:

Example 3-10. SPDAQ Scaler to Ring Buffer (I)

```
#include <CScaler.h>
#include <stdint.h>


class CSIS3820;

class MyScaler : public CScaler
{
private:
  CSIS3820* m_pScaler;
public:
  MyScaler(uint32_t base, unsigned vmeCrate=0);
  virtual ~MyScaler();

  void initialize();
  std::vector<uint32_t> read();
  void clear();

};
```

Not really much to point out here. The method names are all now lower case, the `size` method has bee removed, and the signature of the `read` method has been modified to return a vector of the data read.

The implementation really only differs in the `read` method and even there only trivially. Confining ourselves to that code fragment from the port to the RingDaq we see:

Example 3-11. SPDAQ Scaler to Ring Buffer (II)

```
...
std::vector<uint32_t>
MyScaler:: read()             ❶
{
  uint32_t              channels[32];
  std::vector<uint32_t> scalers; ❷

  m_pScaler->LatchAndRead(reinterpret_cast<unsigned long*>(channels));
  scalers.insert(scalers.end(), channels, channels + 32);

  return scalers;    ❸
}
...
```

❶ The `read` method now returns a vector rather than accepting a reference to one as a parameter.

❷ We have declared a local vector into which the data can be moved. That vector will be returned. Doing this makes the remainder of the code almost identical to the previous code.

❸ Almost identical that is. The vector we fill must be returned to the caller as the function result.

Once we have ported our scalers we need to register them with the `Skeleton.cpp`. This is done in a manner that is identical to the way it is done for production readout code in SPDAQ:

Example 3-12. Registering scaler objects with RingDaq

```
...
#include "MyScaler.h"  ❶
...
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;    ❷
  t.tv_nsec = 0;
  CTimedTrigger* pTrigger = new CTimedTrigger(t);
  pExperiment->setScalerTrigger(pTrigger);

  // Create and add your scaler modules here.

  pExperiment->AddScalerModule(new MyScaler(0x80000000)); ❸

}
...
```

❶ Includes the `MyScaler.h` header so that the class can be instantiated into an object which will be registered as a scaler.

❷ This is the number of seconds between scaler readouts.

❸ Adds the scaler to the ordered list of scalers that will be read out in response to the scaler trigger.

3.5. Modifying the `Makefile`

Finally any code in addition to `Skeleton.cpp` needs to be compiled and linked into the final Readout program. This is done by modifying the `OBJECTS` `Makefile` macro. For our example, assuming we have been porting files named `MyEventSegment.{cpp,h}` and `MyScaler.{cpp,h}`, this means we'll have a line that look like:

```
OBJECTS=Skeleton.o MyEventSegment.o MyScaler.o
```

Chapter 4. Creating a Readout program from a spectrodaq classic readout program

There are still quite a few Readout programs that use the so-called *Readout Classic* framework. This chapter will show how to adapt these programs to use the RingDaq readout framework.

Here are the things you will need to do:

- Obtain the RingDaq Readout skeleton.
- Modify your `skeleton.cpp` and similar files to change `DAQWordBufferPtr` objects into uint16_t* objects and deal with the fallout that causes.
- Write an adaptor event segment that wraps your skeleton file into a RingDaq `CEventSegment`
- Write an adaptor scaler that wraps skeleton into a RingDaq `CScaler`
- Modify the `Skeleton.cpp` to select the event trigger, and register the event and scaler adaptors.
- Modify the `Makefile` to build the software you need in addtion to the `Skeleton`.

4.1. Obtaining the RingDaq skeleton.

In this section we are going to operate as if an environment variable named `DAQROOT` is defined and points to the top level of the RingDaq distribution. At the time this is being written, at the NSCL this would give `DAQROOT` the value `/usr/opt/daq/10.0`. As time goes on, this directory name may change as version numbers change. If you are not at the NSCL you will need to contact your system administrators about where they installed this software.

The commands below show how to obtain a copy of the readout skeleton for RingDaq:

Example 4-1. Getting the skeleton

```
mkdir myreadout
cd    myreadout
cp $DAQROOT/skeletons/sbs/* .
```

This sequence of unix shell commands creates a new directory named `myreadout`, makes that the current default directory and copies the readout skeleton into that directory.

The readout skeleton constists of the following files:

`Makefile`

Makefile that builds the skeleton

`Seleton.cpp`

Source code for the registration code for the readout framework.

`Skeleton.h`

Header defining the class implemented by `Skeleton.cpp`

If you examine `Skeleton.cpp` you wont' find a `main` function. This is because the readout framework is an application framework. Application frameworks consist of a main program that is written for you and specific ways to register the presence of application specific code that needs to be called at well defined points in the program's execution.

Using an application framework frees you from having to worry about how your code actually interfaces with the data acquisition system, manager run-state transitions, trigger processing and so on. In the next two chapters we will see how to create code that is application specific and how to register it with the framework so that it is called when we want it to be called.

In this section we are going to operate as if an environment variable named `DAQROOT` is defined and points to the top level of the RingDaq distribution. At the time this is being written, at the NSCL this would give `DAQROOT` the value `/usr/opt/daq/10.0`. As time goes on, this directory name may change as version numbers change. If you are not at the NSCL you will need to contact your system administrators about where they installed this software.

The commands above just copy the skeleton, most likely you will want to also bring the source code for your existing readout into this directory as well. Suppose these are located in the directory pointed to by the environment variable `oldrdo`:

Example 4-2. Adding existing readout files:

```
mv Makefile Makefile.ring
(cd $oldro; tar czf - .) | tar xzf -
mv Makefile Makefile.original
mv Makefile.ring Makefile
```

These commands assume you may need to copy a directory tree. First the ringdaq Makefile is saved to `Makefile.ring`. Second the directory tree at `oldrdo` is copied via a tar pipeline. Third the Makefile *this* copied in is saved as `Makefile.original`. Finally the ringdaq Makefile is restored from `Makefile.ring`

4.2. Modifications to skeleton.cpp

This section will provide a guide to the modifications you will need to make to a `skeleton.cpp` and files it depends on to be able to use it with the RingDaq readout framework.

The primary modifications are needed in `readevt` and the set of headers and stem from the fact that `spectrodaq.h` no longer exists as RingDaq by definition does not use spectrodaq, and the fact that therefore `DAQWordBufferPtr` objects also no longer exist. `DAQWordBufferPtr` objects are replaced by ordinary pointers to ordinary memory.

Since a `DAQWordBufferPtr` does a good job of imitating an ordinary pointer, these modifications should have minimal impact on your actual code.

I'm going to show two fairly empty `readevt` functions and what you have to do to them. The first is the 'standard' version, where data are taken into `DAQWordBufferPtr` objects directly, while the second uses the `DAQWordBufferPtr::CopyIn` function and a local buffer to improve the SPDAQ performance.

Let's look at the relevant pieces of the first case. Many of the standard comments have been removed for the sake of brevity, as is the standard body.

Example 4-3. A readout classic `readevt`

```
...
#include <spectrodaq.h>   ❶
...
#include <daqinterface.h> ❷
...
WORD
#ifdef __unix__
readevt (DAQWordBufferPtr& bufpt)  ❸
#else
readevt (WORD* bufpt)
#endif
{
#ifdef __unix__
    DAQWordBufferPtr _sbufpt = bufpt; ❹
#else
    WORD *_sbufpt = bufpt;
#endif
    LOGICAL reject;

    reject   = FALSE;
    {
    // code here that invokes putbufw a bunch of times explicitly or
    // implicitly.
    ...
    }
    IF(reject) return 0;
#ifdef __unix__
    return bufpt.GetIndex() - _sbufpt.GetIndex();  ❺
#else
    return (bufpt - _sbufpt);
#endif
}
```

❶ This line includes the `spectrodaq.h` header file. Among other things this defines `DAQWordBufferPtr`

❷ The `daqinterface.h` provides a mini-api to the readout classic library. This also does not exist in RingDaq.

❸ In `__unix__` operating systems, `readevt` is passed a reference to a `DAQWordBufferPtr`. Data then get stored via that object.

❹ In order to be able to compute and return the event size, the 'pointer' object is saved.

❺ The `DAQWordBufferPtr::GetIndex` method returns the offset of the 'pointer' in the underlying buffer. This line therefore determines how many words have been read by `readevt`

In fact the non `__unix__` version of `readevt` is actually almost correct for the RingDaq. Here's how this code fragment would be modified:

Example 4-4. skeleton.cpp modified for RingDaq

```
... ❶
WORD
readevt (WORD* bufpt)    ❷
{

    WORD *_sbufpt = bufpt;

    LOGICAL reject;

    reject   = FALSE;
    {
     // readout code that uses putbufw etc.
     ...
    }

    IF(reject) return 0;

    return (bufpt - _sbufpt);  ❸
}
```

❶ Both the `spectrodaq.h` and the `daqinterface.h` headers are no longer included.

❷ The parameter signature of `readevt` has been modified to take a WORD* rather than a `DAQWordPtr&` paramter. putbufl and its compatriots are pre-processor macros that only require that the `bufpt` variable be in scope and have pointer-like semantics.

❸ Ordinary pointer subtraction can now be used to compute the number of words that were put in the buffer.

Now lets look at the case where `readevt` reads data into a local buffer and then uses `DAQWordBuferPtr::CopyIn` to transfer it to the spectrodaq buffer. Typically `readevt` functions have the following form:

Example 4-5. Classic readout using CopyIn

```
static WORD localBuffer[8192];    ❶
...
WORD
#ifdef __unix__
readevt (DAQWordBufferPtr& bufpt)
#else
readevt (WORD* bufpt)
#endif
{
#ifdef __unix__
    DAQWordBufferPtr _sbufpt = bufpt;
#else
    WORD *_sbufpt = bufpt;
#endif
    LOGICAL reject;

    reject   = FALSE;
    {
        WORD* localbufpt = localbuffer;  ❷
    // code here that invokes localputbufw a bunch of times explicitly or
    // implicitly. putting data into localbuffer.
    ...
        int nWords = localbufpt - localBuffer;  ❸
        bufpt.CopyIn(localBuffer, 0, nWords);   ❹
        bufpt += nWords;                        ❺
    }


    IF(reject) return 0;
#ifdef __unix__
    return bufpt.GetIndex() - _sbufpt.GetIndex();
#else
    return (bufpt - _sbufpt);
#endif
}
```

Key features of this scheme are:

❶ `readevt` functions that follow this pattern declare a local buffer into which data are first read.

❷ A pointer is then created to that local buffer and replacements for `putbufw` usually named something like `localputbufw` are used to put data into this local buffer.

❸ Once readout has been completed, ordinary pointer arithmetic is used to determine how many words have been loaded into the buffer.

❹ The data read are then transfered to the spetrodaq buffer using the `DAQWordBufferPtr::CopyIn` method.

❺ Finally the `DAQWordBufferPtr` object is advanced so that the size calculation done by the framework code is done correctly (or in some cases `nWords` is simply returned at that point).

With RingDaq, since `readevt` is recdeiving an ordinary pointer parameter, there are no efficiency gains to be had by using a local buffer. In order to avoid having to recast all the readout code in terms of e.g. `putbufw` the following trick can be used:

Example 4-6. Converting Classic readout with local buffers to RingDaq

```
❶
#include <stdint.h>      ❷
...
readevt (uint16_t bufpt)
    WORD *_sbufpt = bufpt;
    LOGICAL reject;

    reject   = FALSE;
    {
        WORD* localbufpt = bufpt;     ❸
    // code here that invokes localputbufw a bunch of times explicitly or
    // implicitly. putting data into localbuffer.
    ...
        bufpt = localbuffer;    ❹
    }



    IF(reject) return 0;
    return (bufpt - _sbufpt);
}
```

❷ This header defines integers of known bit widths such as uint16_t an unsigned integer that is guaranteed to occupy exactly 16 bits of storage.

❶ As discussed before the example, we don't need the local buffer any more.

❸ The `localbufpt` is just initialized to point to the buffer passed in to `readevt` this allows macros like `localputbufw` to function properly without source code modification.

❹ Setting `bufpt` to the value of `localbufpt` is all that is then needed to make the size computation performed by the framework code function correctly.

4.3. Writing the adaptors.

Now that we have a `readevt` function that is divorced from Spectrodaq we need to adapt this to the RingDaq readout framework. The RingDaq readout framework uses a pair of base classes that separate event and scaler readout. We will be writing derived classes that simply delegate their functionality to the functions implemented in the `skeleton.cpp` file. If you are familiar with using the `CTraditionalReadoutSegment` and `CTraditionalScalerSegment` classes from the SPDAQ production readout framework to adapt it to classical readout code, you will already be familiar with this concept.

4.3.1. Wrapping `skeleton.cpp` in a `CEventSegment`

The RingDaq readout framework builds up its response to an event trigger in terms of event segments. Event segments can be simple (`CEventSegment` derived objects), or they can be composed of other event segments (`CCompoundEventSegment`). Our job in this section is going to be to build a `CEventSegment` that will wrap the event related functions in `skeleton.cpp`.

Let's start by comparing the functions in `skeleton.cpp` and the related methods in `CEventSegment`.

Our modified `skeleton.cpp` provides three functions that are involved in physics event processing:

```
void initevt(void);
```

```
void clearevt(void);
```

```
void readevt( uint16_t* bufpt );
```

Where these functions perform the following operations

`initevt`

Called to perform one-time initialization as data taking begins (both when the run begins and when it resumes).

`clearevt`

Clears digitizers so that they can accept additional triggers. This is called just after `initevt` as well as after each event is read.

`readevt`

Called to read an event in response to an event trigger. The parameter *`bufpt`* is a pointer to storage into which the event data must be placed.

By contrast, the `CEventSegment` provides the following methods:

```
  void initialize();
  void clear();
  void disable();
  size_t read(void*  pBuffer, size_t  maxwords);
```

Of these methods, only `read` must be implemented. If not implemented the base class provides a method that does nothing. These methods are used by the RingDaq readout framework as follows:

`initialize`

Performs the same sort of initialization `initevt` performs, but only on the devices managed by this event segment.

`clear`

Similarly analagous to `clearevt`

`disable`

This method has no corresponding function in `skeleton.cpp`. It is called at the end of data taking and can be used to do any shutdown tasks that may be required to disable devices.

The preceeding list implies that we should write an event segment in which `initialize` calls `initevt`, `clear` calls `clearevt`, `read` does some adaptation and calls `readevt` and `disable` is not implemented.

The header for this sort of event segment looks like:

Example 4-7. Header for event segment adapator to readout classic

```
#include <config.h>
#include <CEventSegment.h>    ❶

class CTraditionalEventSegment : public CEventSegment ❷
{
public:
  void initialize();
  void clear();
  size_t read(void* pBuffer, size_t maxwords);
};
```

❶ Includes the `CEventSegment` header which is needed to build a derived class.

❷ Our event segment is derived from the `CEventSegment` base class.

Now let's look at the implementation and discuss how that adapts to the functions in `skeleton.cpp` For the most part this is pretty simple as well:

Example 4-8. Implementation for the event segment adaptor to readout classic

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


typedef int16_t WORD;      ❶

extern void initevt ();    ❷
extern void clearevt ();
extern WORD readevt (WORD* bufpt);


❸

void
CTraditionalEventSegment::initialize()
{
  ::initevt();
}
void
CTraditionalEventSegment::clear()
{
  ::clearevt();
}

size_t
CTraditionalEventSegment::read(void* pBuffer, size_t maxwords)
{
  WORD* p = reinterpret_cast<WORD*>(pBuffer);  ❹

  size_t nWords = ::readevt(p);                        ❺
  if (nWords > maxwords) {
    throw std::string("readevt read more than maxwords of data"); ❻
  }
  return nWords;                                       ❼
}
```

For the most part this is very straightforward, and similar to the SPDAQ production readout wrapper class for classic event segments.

❶ Without going through the trouble of defining a specific camac controller implementation, this is the simplest way to make the WORD data type known. WORD is intended to be a 16 bit value. `stdint.h` defines types like uint16_t.

❷ Defines the functions in `skeleton.cpp` we will call as external. It would also be possible to define a `skeleton.h` header and include that instead.

❸ `initialize` and `clear` are trivial delegations to the corresponding `skeleton.cpp` functions.

❹ `read` and `readevt` have a slight impedance match in their argument signatures that needs to be dealt with. This line creates a pointer `p` of type WORD* so that we can pass the correct pointer type to `readevt`.

❺ This calls `readevt` funtion saving the number of words returned.

❻ If the number of words returned is larger than can be accomodated by the buffer we throw an exception. Exceptions of this sort are reported by the readout framework after which the program aborts. In this case, the reported failure likely results in a buffer overrun, so that is the correct action.

❼ If all goes well the number of words is returned to the caller.

4.3.2. Wrapping `skeleton.cpp` in a `CScaler`

We must also write an adaptor taht wraps the scaler parts of `skeleton.cpp` in a `CScaler`. This too is relatively straightforward. Let's once more start by comparing the two software interfaces:

The functions the `skeleton.cpp` uses to manage the scaler readout are:

Scaler interface to `skeleton.cpp`

`void`**`iniscl`**`(void);`

`void `**`clrscl`**`(void);`

`UIINT16 `**`readscl`**`( `*`UINT32* buffer`*` ,  `*`int numscalers`*` );`

`iniscl`

Called to perform run start initialization of the scalers being managed.

`clrscl`

Called prior to the start of run and after each scaler readout. This function is supposed to clear all scaler counters.

`CScaler` methods

```
  void initialize();
  void clear();
  void disable();
  std::vector<uint32_t> read();
```

`initialize`

This method is completely analagous to `iniscl`.

`clear`

This method is completely analaogous to `clrscl`

`disable`

This is called as data taking is shut-down. Any end-run clean up actions can be performed here. This method has no corresponding entry point in `skeleton.cpp` and therefore need not be implemented in wrapper.

`read`

This method is called to read the scalers managed by a `CScaler` object. Unlike the `readsc` function `CScaler` objects are assumed to be managing some fixed set of scalers, and therefore know how many scalers they will read. As we will see there are two strategies you can follow for adapting to this difference.

The `read` method returns an std::vector<uint32_t> that contains the scaler data it has read.

From this comparison we can see it's pretty trivial to wrap `iniscl` and `clrscl`. Here is a header and the first part of the implementation of a wrapper that shows how these functions get trivially wrapped:

Example 4-9. Scaler adapter header

```
#include <CScaler.h>     ❶

class CTraditionalScaler : public CScaler ❷
{
public:                          ❸
  void initialize();
  void clear();
  std::vector<uint32_t> read();
};
```

❶ In order to derive `CTraditionalScaler` from `CScaler` we need to make the shape of `CScaler` known to the compiler. This is done by including this header.

❷ Declares our class as derived from `CScaler` this deriviation allows `CTraditionalScaler` objects to be registered with the framework as `CScaler` objects.

❸ As discussed above, these are the methods we need to implement. The `disable` method need not be implemented as `CScaler` provides a default implementation that does nothing.

The implementation of the trivial wrappers is shown below along with the front matter of the implementation file.

Example 4-10. Trival methods of the scaler adapter

```
#include <config.h>
#include "CTraditionalScaler.h"


typedef uint16_t UINT16;    ❶
typedef uint32_t UINT32;

extern void iniscl();       ❷
extern void clrscl();
extern UINT16 readscl(UINT32* buffer, int numscalers);

void
CTraditionalScaler::initialize() ❸
{
  ::iniscl();
}

void
CTraditionalScaler::clear()      ❹
{
  ::clrscl();
}
```

❶ The simplest way to get these definitions without pulling in too much of the classic framwework is to make these `typedefs`. The first defines an unsigned 16 bit integer to be the meaning of UINT16 while the second defines an unsigned 32 bit integer to be the meaning of uint32_t. The uintxx_t types are defined in the header `stdint.h` which is part of the current C/C++ standard. Going forward those types should be used rather than the UINTxx types because the `stdint.h` types are required to be correct across all compilers.

❷ This set of statements defines prototypes for the functions we are going to be calling in the `skeleton.cpp` file.

❸ `iniscl` is trivially wrapped by this method.

❹ The `clrscl` function is trivially wrapped by this method.

Wrapping the `readscl` function is a bit trickier. Specifically we have to make some decisions about how to know the number of scalers that will be read by the `readscl` function. We need to do this not only to be able to provide the value back to the function (its second parameter), but also to be able to allocate storage for the buffer into which `readscl` will read its data.

There are several strategies that come to mind:

1. Hard code the number of scalers in `CTraditionalScaler`.
2. Add a function to the `skeleton.cpp` code allowing it to report the number of scalers it will read.
3. Make the scaler count a construtor parameter fo the `CTraditionalScaler` class.
4. Provide some mechanism that allows both the `skeleton.cpp` and the `CTraditionalScaler` code to obtain the number of scalers from some external information (e.g. data file, Tcl Script or environment variable.

In this example we will assume that a function named `numScalers` has been added to the `skeleton.cpp` file that reports the number of scaler channesl that will be read. We leave it to you to determine how that function knows this number.

Example 4-11. Adapting the scaler readout

```
...
extern size_t numScalers();     ❶
...
std::vector<uint32_t>
CTraditionalScaler::read()
{
  size_t nChannels = ::numScalers();  ❷
  uint32_t scalerBuffer[nChannels];
  std::vector<uint32_t> result;

  readscl(scalerBuffer, nChannels);    ❸
  for (int i =0; i < nChannels; i++) {
    result.push_back(scalerBuffer[i]);  ❹
  }
  return result;                        ❺
}
```

❶ We have assumed the existence of a function in the `skeleton.cpp` that can provide the number of scaler channels. This declares it so that the compiler will allow us to write a call to it.

❷ These three lines determine how many scaler channels will be read, allocate an ordinary buffer for them and an stl vector to hold our function result.

❸ Next the scalers are read into the ordinary buffer.

❹ This loop transfers the scaler channel data into the vector.

❺ Returns the vector as promised by our method interface

4.4. Modifications to `Skeleton.cpp`

Up until now we have built code to wrap the `skeleton.cpp` in classes that are compatibile with the RingDaq readout framework. The readout framework also requires that we:

1. Select (or write and register) an event trigger.
2. Register an instance of our `CTraditionalEventSegment` as an event segment so that it can respond to the event trigger.
3. Register an instacne of our `CTraditionalScaler` as a scaler so that it can respond to scaler triggers.

The actions above are all done by editing the `Skeleton.cpp` (note the capital S).

Lets first take up the event trigger. In the classical readout framework this is selected at compile time by defining a preprocessor symbol, or by creating and registering a replacement trigger. For the RingDaq readout framework, you must select a trigger in your software. While this is a bit more burdensome, it is more flexible.

Triggering in the RingDaq framework is divided into two parts, the trigger itself, and dead-time management. These are represented by a class descended from `CTrigger` and another class descended from `CBusy`. Objects of these classes can be mixed. It is anticpated that there may be cases where a specific bit of trigger hardware cannot also do deadtime management.

The readout framework supplies trigger/busy classes for the CAEN V262 I/O register and the CAEN V977 coincidence register/latch. In addition it is pretty easy to create new trigger classes and to use them to trigger your readout. The example below selects the CAEN V262 as the trigger and busy manager. The signals used are identical to those used by the SPDAQ frameworks.

Example 4-12. Sepcifying the trigger/busy

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

#include "CCAENEventSegment.h"


#include "CSIS3820Scaler.h"

#include <CCAENV262Trigger.h>   ❶
#include <CCAENV262Busy.h>

...
void
Skeleton::SetupReadout(CExperiment* pExperiment)
```

```
{
...
  pExperiment->EstablishTrigger(new CCAENV262Trigger(0x444400, 0) ); ❷
  pExperiment->EstablishBusy(new CCAENV262Busy(0x444400, 0));        ❸

...
}
```

❶ This header and the next define the `CV262Trigger` and `CV262Busy` classes which we will be using as trigger and busy classes respectively.

❷ This line of code creates a new `CV262Trigger` object for a module with base address of `0x444400` in VME crate 0. This is the traditional location of this module in the NSCL DAQ. The all to the `EstablishTrigger` method of the `CExperiment` object makes this trigger module the experiment event trigger.

❸ Similarly, this line creates a `CV262Busy` object at the same VME base address and establishes it as the module that will handle and maintain the program's busy state.

Now lets register the a `CTraditionalEventSegment` to respond to the trigger. The Readout framework allows you to register any number of event segments. The segments are processed in the order in which you register them, much the same way SpecTcl event processors work.

Example 4-13. Registering a `CTraditionalEventSegment`

```
...
#include "CTraditionalEventSegment.h"  ❶
...
void
Skeleton::SetupReadout(CExperiment* pExperiment)
{
  CReadoutMain::SetupReadout(pExperiment);

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

  pExperiment->EstablishTrigger(new CCAENV262Trigger(0x444400, 0));
  pExperiment->EstablishBusy(new CCAENV262Busy(0x444400, 0));

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

  pExperiment->AddEventSegment(new CTraditionalEventSegment); ❷

}
```

❶ Includes the header for the event segment we wrote to wrap `skeleton.cpp`. This declares the class to the compiler and therefore makes references to it usable later.

❷ Registers the event segment by creating one dynamically and passing the pointer to it to `CExperiment::AddEventSegment`. Since `Skeleton::SetupReadout` only is called once for the lifetime of the program, it is not a memory leak to not provide a way to destroy the pointer.

The code to register the scaler wrapper is about the same, however it is added to `Skeleton::SetupScalers`.

Example 4-14. Registering the scaler adapter with readout

```
...
#include "CTraditionalScaler.h"  ❶
using namespace std;             ❷
...
void
Skeleton::SetupScalers(CExperiment* pExperiment)
{
  CReadoutMain::SetupScalers(pExperiment);      // Establishes the default scale
r trigger.

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

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

  // Create and add your scaler modules here.

  pExperiment->AddScalerModule(new CTraditionalScaler); ❹

}
```

❶ As before we must include the header for our wrapper so that we can refere to and use the `CTraditionalScaler` class later in the code.

❷ If you are used to using classes and objects that are in the `std` namespace in C++ without prefixing them with `std::` you should inlude this line. It incorporates such classes/objects, like `std::string` or `std::cerr` etc, into the compilers unqualified name search path so that the explicit use of the `std::` prefix is not required to use those classes and objects.

❸ This code comes pre-packaged with the `Skeleton.cpp` file. It sets up a scaler readout trigger to fire every two seconds when the run is active. Scaler triggers, like event triggers are completely replaceable. If you want a different scaler readout interval, change the `2` to the number of seconds between readouts.

While the `CTimedTrigger` supports trigger intervals that are not whole seconds, the time resolution of the run time offset in the scaler event currently only supports whole seconds. Therefore leave the *`t.tv_nsec`* (nanoseconds) field set to zero.

❹ Registers our scaler module with the readout. As for event segments, any number of scaler modules can be registered. Scaler modules will be read out in the order in which they are registered.

Once more, since the `CSkeleton::SetupScalers` is only called once in the lifetime of the program, no memory leak results from creating the `CTraditionalScaler` object in the way we have.

4.5. Modifying the Makefile

Now that our code is all written, we must ensure it will be compiled into the final executable program. This is a matter of adding the adaptor modules we wrote to the `OBJECTS` makefile macro definition in `Makefile`. The example below shows the resulting definitions assuming that we have named our event readout adaptor `CTraditionalEventSegment.cpp` and our scaler adaptor `CTraditionalScaler.cpp`:

```
OBJECTS=Skeleton.o skeleton.o CTraditionalEventSegment.o \
        CTraditionalScaler.o
```

Naturally you will also need to add any objects for source files you need from your old Readout program to this definition. Do *not* add `ReadoutMain.o` to this list as the new framework is not compatibile with it and does not need it.

Chapter 5. Analyzing ring buffer data with SpecTcl

This chapter describes how to analyze ring buffer data with SpecTcl. As with the Readout, ring buffer data analysis is highly source code compatible. To analyze the data requires two sets of modifications to your SpecTcl:

WARNING

Your SpecTcl must have first been upgraded to version 3.3 before you can analyze ring buffer data in native mode. See, however Appendix C for information about compatibility mode utilities that might allow older SpecTcl versions to analyze ring buffer data.

1. You must alter the scripts you use to connect SpecTcl to the online system to use the correct pipe data source, data source format and URL for RingDaq.
2. You must make some simple changes to your analysis to handle the fact that the event length is now 32 bits and, if you are using documented packets, the packet size fields are 32 bits.

NOTE: You will not need to make any changes to your Makefile if you are using SpecTcl version 3.3 or greater as this version of SpecTcl already knows how to handle data buffers containing ring items (via the `-format ring` switch on the **attach** command).

This section takes a simple example and shows how to perform these modifications. We use very generic SpecTcl analysis code that has the actual analysis abstracted away. We also assume the existence of an `attachOnline` Tcl **proc** that both need to be modified.

5.1. Event processor modifications

Let's start by looking at the boilerplate of a typical event processor for SPDAQ shown in the example below.

Example 5-1. SpecTcl event processor boilerplate

```
#include <config.h>
#include "MyEvProc.h"
#include <TranslatorPointer.h>    ❶
#include <BufferDecoder.h>
#include <TCLAnalyzer.h>
```

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

  ❷
  TranslatorPointer<UShort_t> p(*(rDecoder.getBufferTranslator()), pEvent);
  UShort_t  nWords = *p++;                                         ❸
  CTclAnalyzer&      rAna((CTclAnalyzer&)rAnalyzer); ❹
  rAna.SetEventSize(nWords*sizeof(UShort_t)); // Set event size. ❺

  // Here we would have code that unpacked pEvent into rEvent.

  //

  return kfTRUE;    ❻
}
```

❶ These includes are typically required by event processors. `MyEvProc.h` is the header for this event processor class. All of the other headers are standard SpecTcl class definitions.

❷ SpecTcl supports transparent byte order conversions for integer data types in the event. It does this via a pointer-like object called a `TranslatorPointer`. The next line creates a translator pointer to access the events as unsigned shorts (Ushort_t).

❸ The SPDAQ data acquisition readout frameworks precede each event with a 16 bit word count (count of 16 bit items in the event). This line extract that word count from the event.

❹ At least one event processor in the event processing pipeline is required to inform SpecTcl of the number of bytes in the event. This allows SpecTcl to locate the next event in the data stream.

Since `operator()` return value is already used to indicate success or failure of the event processing pipeline, this is done by calling a method in the `CTCLAnalyzer` which is an object that controls the overall flow of' analysis in SpecTcl. This line casts the generic `CAnalyzer` into the correclt type of analyzer.

❺ Sets the event size for SpecTcl.

❻ Returnging `kfTRUE` tells the analyzer that called this event processor that analysis was successful and to procede to the next stage of the event pipeline or to pass the generated event to the histogramming part of SpecTcl.

To adapt this event processor to RingDaq requires that we modify the boilerplate code slightly. The modification is required because the event size in RingDaq has been widened to a 32 bit unsigned integer to allow for larger events (e.g. waveform digitizers).

The example below shows the boiler plate modified to take into account this change. For the sake of brevity only the relevent code fragment is shown:

Example 5-2. SpecTcl event processor boilerplate for RingDAQ

```
 ...
    ❶
  TranslatorPointer<ULong_t> pwc(*(rDecoder.getBufferTranslator()), pEvent);
  ULong_t  nWords = *pwc++;       ❷
  TranslatorPointer<UShort_t> p(pwc);  ❸
  CTclAnalyzer&      rAna((CTclAnalyzer&)rAnalyzer);
  rAna.SetEventSize(nWords*sizeof(UShort_t)); // Set event size.
...
```

❶ Obtains a translator pointer as before, but in this case a longword translator pointer is used so the widened RingDaq word count can be extracted.

❷ Extracting the word count in this way leaves `pwc` pointing to the event body.

❸ Converts the longword translating pointer to a translator pointer to UShort_t as before. Choosing the target to be `p` should allow all the remaniing event processor code to run un-modified.

Before leaving this topic, let's look at one more issue; packets. Depending on how you build packets you may need to modify the code that gets the size of the packet from each packet. If you create packets manually, or via the packet macros of the classic readout framework, you don't need to make any additional changes to your SpecTcl code.

If you use `CDocumentedPacket`s to create your packets, you will need to modify your analysis of those packets to handle the widened packet size fields of those packets. As with the event, `CDocumentedPacket`s now use a 32 bit unsigned packet size.

Below is a code fragment that extracted the packet size from a SPDAQ documented packets and pointed to the packet body:

```
...
  TranslatorPointer<UShort_t> p(*(rDecoder.getBufferTranslator()), pEvent);
...
   UShort_t packetSize = *p++;
```

Code like this should be changhed to look like:

```
...
  TranslatorPointer<ULong_t> pwc(*(rDecoder.getBufferTranslator()), pEvent);
  ULong_t  nWords = *pwc++;
  TranslatorPointer<UShort_t> p(pwc);
...
```

```
pwc = p;
ULong_t packetSize = *pwc++;
p = pwc;
...
```

The key is to use a TranslatorPointer<ULong_t> to fish the size out of the event, and that assigning translator pointers works as you expect it would.

5.2. Attaching to a ring buffer data source

This section provides guidance on how to attach SpecTcl to a RingDaq ring buffer data source. Each graphical user interface written for SpecTcl has its own methods for attaching to data sources. This makes it a bit tough to talk in generalities. What I will assume for this is the existence of a **proc** named `attachOnline` that receives as a parameter, the name of the host from which data will be taken.

For RingDaq, this proc must:

1. Locate the ringselector executable. ringselector selects data from a ring and send it to stdout. It is the preferred SpecTcl pipe data source for RingDaq. See the comprehensive documentation http://docs.nscl.msu.edu/daq/ringbuffer (http://docs.nscl.msu.edu/daq/ringbuffer) for more information about this application.
2. Construct the correct URL to use as a data source for the ringselector application and use it to construct the correct ringselector command.
3. Issue the **attach** command for a pipe data source specifyig the `-format ring` option.

In the code for ringselector we are going to assume that the version for the ringbuffer data acquisition system is 10.0 or greater. To simplify the search we will assume there are not versions higher than 19.9 and that the 'point' releases are all single digits.

The following fragment of Tcl code locates the top level directory of the highest DAQ version greater than 10.0 at the NSCL:

```
set versions [glob /usr/opt/daq/1\[0-9\].\[0-9\]]  ❶
set versions [lsort -decreasing $versions]        ❷
set highestVersion [lindex $versions 0]           ❸
```

❶ `versions` is set to the list of matching top level daq directories. The glob pattern requires backslash substitutions to allow the range patterns (`[0-9]`) to not be interpreted as command substitutions.

❷ The version directory names are then sorted in high to low order.

❸ The highest version is then the first in the list.

Given a hostname, in the variable `hostname` a URL has to be constructed of the form: `tcp://hostname/username` in order to get data from the correct ring. The host `localhost` will get data from the local ring without making a proxy ring.

The Tcl fragment below will create that url:

```
global tcl_platform                             ❶
set url [join [list tcp: "" $hostname $tcl_platform(user)] /]   ❷
```

❶ The built in global variable `tcl_global` is an array that contains among other things `tcl_global(user)` which is the logged in username. Since we are building the body of a proc, we need to declare it **global** to use it.

❷ The **join** command createsa a single string by joining list elements together with the separator character provided (in this case `/` The empty list element `""` in the list is a way to get the pair of slashes needed between the protocol (`tcp:`) and the hierarchical part.

Putting this all together gives us this:

Example 5-3. Proc to connect SpecTcl to a ring buffer data source

```
 proc attachOnline hostname {
    global tcl_platform

    set versions [glob /usr/opt/daq/1\[0-9\].\[0-9\]]
    set versions [lsort -decreasing $versions]
    set highestVersion [lindex $versions 0]
    set ringHelper [file join $highestVersion bin ringselector]  ❶

    set url [join [list tcp: "" $hostname $tcl_platform(user)] /]

    attach -format ring -pipe \
        $ringHelper --source=$url --sample=PHYSICS_EVENT    ❷
}
```

❶ The **file join** command creates the path to the ringselector program. ringselector is the the `bin` subdirectory of the installation directory tree of the version of the RingDaq we found.

❷ This command does the actual attach. The `-format ring` option tells attach that the data will come in ringbuffer format. You must also use this option when attaching to an event file produced by the RingDaq..

The second line of the command is the ringselector command used to send data to SpecTcl through a pipe. The `--source` option specifies where data comes from. The `--sample` option specifies that data of type `PHYSICS_EVENT` can be sampled. That is when ringselector is getting data from the ring it is allowed to skip physics events if SpecTcl is getting too far behind.

5.2.1. RingDaq event files

In order to read data from event files that are written by RingDaq you need only add `-format pipe` to your **attach** command e.g.:

```
set filename [format run-%04d-00.evt $runNumber]
attach -format -ring -file [file join ~ stagearea complete $filename]
```

Where the fragment above assumes you are opening segment 0 of a run whose run number is in the Tcl variable `runNumber`

Appendix A. Format of Ring bufffer DAQ event files

This information is provided for people who want to write programs to analyze event files written by the RingDaq. This appendix consists mostly of an annotated version of `DataFormat.h` in the `include` directory of a RingDaq installation.

RingDaq event files consist of a stream of *items* Each item has a header that consists of a 32 bit byte count (not word count), and a 32 bit type code for the item. Immediately following the header is the item payload. The type code must have the most significant 16 bits set to `0` this allows you to determine the endian-ness of the data by looking at which word of the type code is non-zero.

The count (which may have to be byte swapped if the type code indicates the byte order of the generating system is different than analyzing system), is self inclusive. Items can therefore be easily skipped.

32767 Types are reserved for use by the RingDaq software while the remaining 32767 types can bge used by user applications with special needs.

`DataTypes.h` defines the type fields as follows:

Example A-1. Item type codes

```
// state change item type codes:

static const uint32_t BEGIN_RUN(1);
static const uint32_t END_RUN(2);
static const uint32_t PAUSE_RUN(3);
static const uint32_t RESUME_RUN(4);

// Documentation item type codes:

static const uint32_t PACKET_TYPES(10);
static const uint32_t MONITORED_VARIABLES(11);

// Scaler data:

static const uint32_t INCREMENTAL_SCALERS(20);

// Physics events:

static const uint32_t PHYSICS_EVENT(30);
static const uint32_t PHYSICS_EVENT_COUNT(31);

// User defined item codes

static const uint32_t FIRST_USER_ITEM_CODE(32768); /* 0x8000 */
```

The payload structure descriptions will provide further information about the meaning of each of these type codes.

The structs below provide definitions used by RingDaq for both the item headers and a generic item:

```
typedef struct _RingItemHeader {
  uint32_t     s_size;
  uint32_t     s_type;
} RingItemHeader, *pRingItemHeader;


typedef struct _RingItem {
  RingItemHeader s_header;
  uint8_t        s_body[1];
} RingItem, *pRingItem;
```

`s_size`

Is the size of the ring item in bytes

`s_type`

Is the item type as described above.

`s_header`

Is the header of the item

`s_body`

Is a placeholder for the item payload.

Normally if you get a RingItem or pRingItem you will cast it to one of the specific ring item types described below. Doing that will provide a detailed break out of the payload.

Run state changes are documented by inserting state change items that have the structure shown below. These have types that are one of `BEGIN_RUN`, `END_RUN`, `PAUSE_RUN` or `RESUME_RUN`. If you receive a pointer to an item of this sort and cast it to a pStateChangeItem, you'll be able to acces the payload fields.

```
typedef struct _StateChangeItem {
  RingItemHeader  s_header;
  uint32_t        s_runNumber;
  uint32_t        s_timeOffset;
  time_t          s_Timestamp;
  char            s_title[TITLE_MAXSIZE+1];
} StateChangeItem, *pStateChangeItem;
```

`s_runNumber`

Is the run number of the run that is undergoing a state transition.

`s_timeOffset`

Is the number of seconds into the run the transition occured. For a begin run, this is by definition 0.

`s_Timestamp`

Is the absolute time at which the run started. This is expressed in the unix time format as the number of seconds since midnight January 1, 1970 GMT.

`s_title`

A null terminated string containing a run title. You should not assume this array has any specific values after the first null.

RingDaq periodically reads scaler data. These result in ring items of type `INCREMENTAL_SCALARS`. The shape of this ring item is shown below:

```
typedef struct _ScalerItem {
  RingItemHeader  s_header;
  uint32_t        s_intervalStartOffset;
  uint32_t        s_intervalEndOffset;
  time_t          s_timestamp;
  uint32_t        s_scalerCount;
  uint32_t        s_scalers[1];
} ScalerItem, *pScalerItem;
```

`s_intervalStartOffset`

The number of seconds into the run at which the interval over which the scalers were accumulated started.

`s_intervalEndOffset`

The number of seconds into the run at which the scaler counting interval ended (when the scalers were read and cleared).

`s_timestamp`

The Unix timestamp at which the scalers were read.

`s_scalerCount`

The number of scalers actually read. Scalers are assumed to be uint32_t devices.

`s_scalers`

Placeholder for an array of *`s_scalerCount`* 32 bit unsigned scaler values.

Two item types have a payload which is just an array of null terminated strings. The contents of these strings depends on the actual item type. These ares stored in TextItem items. The meaning and contents of each string depends on the item type.

`PACKET_TYPES`

> This item contains strings that describe the set of `CDocumentedPackets` that occur within a run. Each string is a colon separated set of fields containing in order the packet id, short packet name, packet description, packet version and the date/time at which the packet was instantiated.

`MONITORED_VARIABLES`

> This item contains monitored variable values. A monitored variable is a Tcl variable whose value is periodically written to the event file. The value of one of these variables might change either because the readout software changed it or, more commonly because the server component of the readout framework was enabled and a client poked a new value to the variable (e.g. as in EPICS monitoring).
>
> Each string consists of a Tcl script fragment that, if executed in an interpreter will set that specified variable to its value at the time the item was emitted. For example **set something something-else**.
>
> Strings are null terminated.

The complete format of the TextItem ring item is:

```
typedef struct _TextItem {
  RingItemHeader s_header;
  uint32_t       s_timeOffset;
  time_t         s_timestamp;
  uint32_t       s_stringCount;
  char           s_strings[1];
} TextItem, *pTextItem;
```

`s_timeOffset`

> Number of seconds into the run at which this item was emitted

`s_timestamp`

> Unix timestamp for the absolute time at which this item was emitted.

`s_stringCount`

> The number of strings that are in the payload.

`s_strings`

> Placeholder for the *`s_stringCount`* null terminated fields.

Each trigger produces a `PHYSICS_EVENT` item. The format of the event is assumed to be completely up to the readout framework (it is envisioned that in the future there will be other readout frameworks available). The PhysicsEventItem, therefore is identical in shape to the RingItem:

```
typedef struct _PhysicsEventItem {
  RingItemHeader s_header;
  uint16_t       s_body[1];
} PhysicsEventItem, *pPhysicsEventItem;
```

From time to time a PhysicsEventCount item is emitted. This item allows the data rate to be easily computed (accepted triggers/sec). It also allows sampling consumers such as SpecTcl to compute the fraction of the data they have analyzed. These items are of type `PHYSICS_EVENT_COUNT` and have the following shape:

```
typedef struct __PhysicsEventCountItem {
  RingItemHeader s_header;
  uint32_t       s_timeOffset;
  time_t         s_timestamp;
  uint64_t       s_eventCount;  /* Maybe 4Gevents is too small ;-) */
} PhysicsEventCountItem, *pPhysicsEventCountItem;
```

`s_timeOffset`

Seconds into the run at which this item was emitted.

`s_timestamp`

Unix timestamp for the absolute time at which this item was emitted.

`s_eventCount`

Number of triggers accepted so far in the run.

Appendix B. User written triggers

The readout framework has explicity support for user written triggers. This appendix will describe that support and provide a simple example.

A *trigger* is a class/object that is polled to determine if a condition has occured. Two triggers are registered by the application programmer. An event trigger, which determines when the `read` method of the event segments are invoked, and a scaler trigger, which determines when scalers get read/cleared.

Closely associated with triggers are Busy objects which allow the computer to report when it is not able to accept a new trigger. Busy objects are only used with the event trigger.

All triggers are required to extend the `CEventTrigger` (event in this case means that some external event occured not that a physics event of interest has been detected). The `CEventTrigger` provides the following public interface to clients:

```
virtual void setup();
virtual void teardown();
virtual  = 0 bool operator()();
```

`setup`

Called as the run is starting. This allows the trigger code to do any hardware setup required to initialize the trigger. If not declared/implemented, the base class provides a no-op default implementation.

`teardown`

Called as a run is halting (pausing as well as ending). This allows for the trigger class to do any required shutdown of the trigger hardware. If not declared/implemented, the base class provides a no-op default implementation.

`operator()`

This is called when the framework is able to accept a trigger. The method is expected to return `true` if a trigger is present, and `false` if not. The method should not poll. That is the function of the framework. The method should determine, as quicly as possible the state of the trigger and return the appropriate value. This method is mandatory and is pure virtual in the base class.

Without comment I supply a trigger class that can trigger the DAQ when a CAEN V775/782/792/865 module shows data present.

Example B-1. CAEN data ready trigger header

```
#ifndef __CEVENTTRIGGER_H
#include <CEventTrigger.h>
#endif
```

```
class CAENcard;         /* Forward class definition */

/*!
  This class is a trigger class that will indicate a trigger if a specific
  CAEN32 module has data.
*/

class CCAENRdyTrigger : public CEventTrigger
{
private:
  CAENcard*   m_pModule;

public:
  CCAENRdyTrigger(CAENcard* pSegment);

  virtual bool operator()();    // Only method we need to implement.
};
```

Example B-2. CAEN data ready trigger implementation

```
#include <config.h>
#include "CCAENRdyTrigger.h"
#include <CAENcard.h>



CCAENRdyTrigger::CCAENRdyTrigger(CCAENCard* pSegment) :
  m_pModule(pSegment)
{}

bool
CCAENRdyTrigger::operator()()
{
  return m_pModule->dataPresent();
}
```

Note that by hooking the command bus for a set of these modules together, you coudl use `gdataPresent` as the trigger conditions and then the trigger would be true if any of the modules had data.

Appendix C. Compatibility mode utilities

In order to facilitate migration to the new ringdaq software, several utilities have been written that allow spdaq client software to attach to ring buffer data with little or no modification. This appendix briefly describes these utilities and provides some examples of their use.

The utilities address three areas of compatibility:

- Event file format.
- SpecTcl pipe data sources
- Hoisting data to systems that are not running nscldaq software

Event file format. The eventlog-compat script provides an event logger that the ReadoutShell GUI can use to log event data in SPDAQ compatible format. Readout Shell has been enhanced so that alternative event loggers can be used. This combination allows you to run experiments recording data directly in SPDAQ compatible format.

When Readout shell starts, it looks for the environment variable `EVENTLOGGER` if this is defined it is assumed to be the full absolute path to an event logging program which will be used instead of the default eventlog. The example belows shows how to make use of this to write event data directly in SPDAQ format

Example C-1. Writing event data in SPDAQ format

```
export EVENTLOGGER=/usr/opt/daq/10.0/bin/eventlog-compat
```

Piping SPDAQ formatted data to SpecTcl. The spectcldaq provides a SpecTcl pipe data source that sends data to SpecTcl in SPDAQ format. This allows you to attach unmodified SpecTcl applications to the ring buffer DAQ system.

The example below is a Tcl **proc** that allows you to make use of this facility:

Example C-2. Piping SPDAQ formatted event data to SpecTcl

```
proc attachOnline host {
    global tcl_platform
    global env

    set user    $tcl_platform(user)
    set daqroot $env(DAQROOT)
    set spectcldaq [file join $daqroot bin spectcldaq]

    attach -pipe $spectcldaq tcp://$host/$user
    start
}
```

The software assumes the environment variable `DAQROOT` is defined and is the top level directory of the ringdaq installation (e.g. `/usr/opt/daq/10.0/bin`). The script further assumes that the readout source is putting data into its default ring. Note that as with Spectrodaq, using `localhost` when you want data from the same system is preferred over specifying the hostname.

Serving SPDAQ data to non NSCLDAQ systems (e.g. S800 Mac). The spectcldaq.server script implements a server that allows connecting systems to get SPDAQ formatted data across a TCP/IP client/server connection. This can be used to send data to the S800 Mac SpecTcl diagnostics.

spectcldaq.server must be parameterized with the URL of the ring buffer into which Readout is putting data as well as the the port on which it listens. Any connecting client will receive data in SPDAQ fromat along its socket until it disconnects.

The next pair of examples shows how to start the server in the spdaq system that provides data listening on port 1100, and how to provide a pipe data source to SpecTcl using the server.

Example C-3. Running spectcldaq.server in an Spdaq system.

```
$DAQROOT/bin/spectcldaq.server tcp://localhost/‘whoami‘ 1100
```

The example above assumes `DAQROOT` points to the base of the ringdaq directory tree (.e.g **export DAQROOT=/usr/opt/daq/10.0**), and that the Readout software is putting data into its default ring. Note that when accessing rings that are local to the machine the use of `localhost` is preferred in the ring url.

Example C-4. SpecTcl pipe data source using spectcldaq.server

```
proc attnet {host port} {
    attach -pipe netcat $host $port
    start
}
```

This **proc** makes use of the netcat utility which connects to a TCP/IP host/port pair and outputs data from that connection on its stdout. netcat is available on all Linux systems and is avaialble in the Darwin ports repository for OS-X, (http://www.darwinports.info/ports/net/netcat.html (http://www.darwinports.info/ports/net/netcat.html)) as well as being available in source form under the GPL (http://netcat.sourceforge.net/).