Using NCSL DAQ Software to Readout a Phillips 1784 ADC or 1786 TDC

Timothy Hoagland

January 21, 2005
Abstract

This paper’s purpose is to assist the reader in setting up and reading out data from a Phillips 1784 ADC or 1786 TDC. It covers what electronics will be needed and how they should be setup. It will show what software modifications need to be done and gives sample code to use.

This document is meant to serve people with a wide range of experience, however some previous knowledge is needed. This paper assumes that you are at least somewhat familiar with Linux since the DAQ software runs on a Linux box. It also assumes that you a little familiar with C++. Finally it will be helpful if you know how to use an oscilloscope, as that will be needed to setup your electronics.

All the code is available at:
http://docs/daq/samples/PhillipsCAMAC/PhillipsCAMAC.zip
Contents

1 A Brief Description of the Phillips 1784 and 1786 2

2 Minimum Electronics Setup 2
   2.1 Phillips 1786 TDC in common start mode .............. 2
   2.2 Phillips 1784 ADC ............................................ 4
   2.3 CAMAC branch controllers .................................... 4

3 Modifying Readout 5
   3.1 Creating MyEventSegment ................................. 5
   3.2 Modifying Skeleton.cpp ................................. 5
   3.3 Modifying Makefile ........................................... 6
   3.4 Testing Readout .................................................. 7

4 Modify SpecTcl 7
   4.1 Creating MyEventProcessor ............................... 8
   4.2 Modifying MySpecTclApp.cpp ............................... 8
       4.2.1 Making SpecTcl ........................................... 9
   4.3 Writing the SpecTcl Script ............................... 9

5 Testing 9

6 Using The WIENER VC32 10

7 More information 10

8 Code Examples 11
   8.1 MyEventSegment.h ........................................... 11
   8.2 MyEventSegment.cpp ........................................ 11
   8.3 MyEventProcessor.h ......................................... 13
   8.4 MyEventProcessor.cpp ..................................... 13

List of Figures

1 Electronics setup for 1786 (common start) ................. 3
2 Electronics setup for 1784 .................................... 4
1 A Brief Description of the Phillips 1784 and 1786

Both the Phillips 1784 and 1786 are 12 bit CAMAC based digitizers. They share many internal electronics including the CAMAC interface, this means that the same piece of code can be used to readout either module. The 1784 ADC has a dynamic range of 4V and can accept gates up to 50 microseconds long. The 1786 TDC can be run in either common stop or common start and has a full range of 800ns, although that could be extended through a factory option. Both modules have a conversion and processing time less than 7.2 microseconds.

This summary of the module is incomplete, please read the manual for complete information.

2 Minimum Electronics Setup

A minimum electronics setup will allow you to test your software when you are done. An example Setup for the 1786 and 1784 are shown in sections 2.1 and 2.2 respectively. Brief discussions of the operation and use of discriminators and gate and delay generators are available at http://docs.nscl.msu.edu/daq/samples

2.1 Phillips 1786 TDC in common start mode

Figure 1 shows a small test setup for the Phillips 1786. This setup shows the module being used in common start mode. To set your module in common start the internal jumpers for each channel be set as described in the users manual.

For the purpose of this example the delay on channel two of the discriminator was set to about 200ns. The jumpers on the back of the 1786 were set for a full range of 800ns and a conversion delay of 750ns (see manual for jumper details).

The CAEN V262 in the setup is used to trigger the VME readout.
Figure 1: An Example of an electronics setup for the Phillips 1786 in common start mode.
Figure 2: An Example of an electronics setup for the Phillips 1784.

### 2.2 Phillips 1784 ADC

Figure 2 shows a small test setup for the Phillips 1784. For the purpose of this example the gate is set to about 5 microseconds. The jumpers on the back of the 1784 were set for a conversion delay of 750ns (see manual for jumper details).

The CAEN V262 in the setup is used to trigger the VME readout.

### 2.3 CAMAC branch controllers

Since the CAMAC crate will be read out via a VME crate it is necessary to link the two crates with CAMAC branch highway. The NSCL uses two kinds of VME-CAMAC controllers.

The first is the CES CBD8210. This system consists of a VME module, a double wide CAMAC controller and a single width CAMAC termination module. The modules are connected with specially designed branch cables. Using this system up to 7 CAMAC crates can be controlled through a single VME module by connecting the CAMAC controllers in a daisy chain;
the termination module should be at the end of the chain. The cabling is expensive and difficult to work with.

The current choice for VME-CAMAC controllers at the NSCL is the WIENER VC32/CC32. This system requires one VME module for each CAMAC crate. The double wide CAMAC controller incorporates a CAMAC bus analyzer on the front panel. A standard SCSI-II cable is used to connect the VME module and the CAMAC controller.

In this example the CESCAMAC controller will be used because it was readily available for testing at the time of this writing. For information on what software changes are needed to use the WIENER VC32 controller see section 6 on page 10

3 Modifying Readout

To obtain a copy of the Readout skeleton type:

```bash
mkdir -p ~/experiment/readout
cd ~/experiment/readout
cp /usr/opt/daq/pReadoutSkeleton/* .
```

This will create a directory named /experiment/readout and then copy four files to the new directory.

3.1 Creating MyEventSegment

In order to read data out of any module we first have to tell the software what modules we have, where they are and how to read them. We will accomplish this by creating a class called MyEventSegment. The class will define four functions and four private variables, needed to define our Phillips digitizer. The code for the header file and implementation file can be found in sections 8.1 and 8.2, respectively, beginning on page 11)

3.2 Modifying Skeleton.cpp

After defining MyEventSegment we are ready to modify Skeleton.cpp so that it knows to include our new class. After including your new header file at the top of the file locate the block of code that reads:
3 MODIFYING READOUT

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

    // Insert your code below this comment.

    rExperiment.AddEventSegment(new MySegment);
}

Replace that block of code with:

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

    // Insert your code below this comment.

    rExperiment.AddEventSegment(new MyEventSegment(12,2,1,1));
}

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

3.3 Modifying Makefile

We are now ready to compile our code but first we will have to make two modifications to the Makefile. Open Makefile and on the objects line add MyEventSegment.o. The line should look like this when you’re done.

Objects=Skeleton.o MyEventSegment.o
Then add -DCESCAMAC as a compilation switch. It should look like this:

\texttt{USERCXXFLAGS= -DCESCAMAC}

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

\subsection*{3.4 Testing Readout}

Type ”Readout” to start the readout program, typing ”begin” at the prompt will start a data run. In a separate shell type ”Bufdump”. This program will dump the data to the screen where we can look at it. An example of what you might see is:

\begin{verbatim}
--------------- Event (first Event) ------------------
  Header:
  4096 1 -16830 0 92 0 2220 0 0 0 5 258 772 258 0 0
  Event:
  5(10) words of data
  5 4 c 0 3e4
\end{verbatim}

The words scrolling on the screen are in hexadecimal representation. The first word in the event data is an inclusive word count of the number of words in the packet. The third word should be the packet Id you used earlier. If both of these numbers are correct, you are ready to continue to setting up SpecTcl.

\section*{4 Modify SpecTcl}

Begin by obtaining a copy of the SpecTcl skeleton files by typing:

\texttt{cp /usr/opt/spectcl/current/Skel/*}
4 MODIFY SPECTCL

4.1 Creating MyEventProcessor

In order to decipher the data packet create in MyEventSegment a class second class will be defined. This class, MyEventProcessor, will have three functions and two private variables. The code for the header file and implementation file can be found in sections 8.3 and 8.4, respectively, beginning on page 11.

4.2 Modifying MySpecTclApp.cpp

In order to hook MyEventProcessor into SpecTcl we will need to modify MySpecTclApp.cpp. The first step is to include ”MyEventProcessor.h” in the list of includes. Next define a variable of type MyEventProcessor, with the same Id as we used earlier and a base index for the data, as shown:

\[ \text{MyEventProcessor MyProcessor(100, 12); } \]

Now locate the CreateAnalysisPipeline function that looks like this:

\[
\text{void CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer) }
\]

\[
\text{ { }
\]

\[
\text{ #ifdef WITHF77UNPACKER }
\]

\[
\text{ RegisterEventProcessor(legacyunpacker); }
\]

\[
\text{ #endif }
\]

\[
\text{ RegisterEventProcessor(Stage1); }
\]

\[
\text{ RegisterEventProcessor(Stage2); }
\]

\[
\text{ } }
\]

The values of our base index and our packet Id should be passed here. The base index is the starting array index that the values will be stored in, the Id is the one you used earlier.

\[
\text{void }
\]
5 Testing

CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
  RegisterEventProcessor(MyProcessor);
}

4.2.1 Making SpecTcl

We are now ready to edit our Makefile so that we can compile our program. Find the line that reads:

OBJECTS=MySpecTclApp.o

and replace it with:

OBJECTS=MySpecTclApp.o MyEventProcessor.o

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

4.3 Writing the SpecTcl Script

The last piece of code we have to write is the spectcl setup script. This the piece of code that tells the software what is available to histogram. Start by creating a file called setup.tcl, which is shown at the end of this document.

5 Testing

We are now ready to test our event processor. Begin by starting Readout and beginning a run. Then start SpecTcl. When SpecTcl starts a window with a command prompt will open type "source setup.tcl" This will create an "attach online" button in the SpecTcl window. Click this attach online button. Follow these steps to see your spectrum:

- Click on the Xamine window
Using The WIENER VC32

The WIENER VC32 can be used in place of the CESCAMAC branch controller. When this module is used a couple of changes are needed in your code. First because of the way the WIENER controller counts, increase the branch number by one in your code (1 becomes 2). The other modification is to change the compilation switch used in the Makefile. Locate the line that reads:

```
USERCXXFLAGS= -DCESCAMAC
```

Replace -DCESCAMAC with -DVC32CAMAC. You are now ready to compile.

More information

This example setup is designed to be quick and easy. It should not be seen a comprehensive source of information on either the Phillips 1784 or 1786. More information is available at: http://docs.nscl.msu.edu/ and should be your first source for help. If that doesn’t help contact daqdocs@nscl.msu.edu.

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

8.1 MyEventSegment.h

```c
#ifndef __MTEVENTSEGMENT_H
#define __MTEVENTSEGMENT_H
#include <spectrodaq.h>
#include <CEventSegment.h>
#include <CDocumentedPacket.h>
#include <camac.h>

// Declares a class derived from CEventSegment
class MyEventSegment : public CEventSegment
{
    private:
    CDocumentedPacket m_MyPacket;
    unsigned int Tag;    // Packet ID Tag
    unsigned int Slot;   // CAMAC slot module is in
    unsigned int Branch; // CAMAC Branch #
    unsigned int Crate;  // CAMAC crate #
    public:
    MyEventSegment(unsigned short tag, int slot, int branch, int crate);
        // Defines packet info, sets private variables
        virtual void Initialize();  // One time Module setup
    virtual void Clear();        // Resets data buffer
    virtual unsigned int MaxSize();
    virtual DAQWordBufferPtr& Read(DAQWordBufferPtr& bufpt);
        // Reads data buffer
};
#endif
```

8.2 MyEventSegment.cpp

```c
#include "MyEventSegment.h"
static char* pPacketVersion = "1.0";
// Packet version - should be changed whenever major changes are made

// constructor set Packet details
MyEventSegment::MyEventSegment(unsigned short tag, int slot, int branch, int crate):
    m_MyPacket(tag, string("My Packet"), string("Sample documented packet"),
    string(pPacketVersion))
{
    Tag = tag;  // Packet Id #
    Slot = slot; // CAMAC slot #
    Branch = branch; // CAMAC branch #
    Crate = crate;  // CAMAC crate #
}
```
// Is called right after the module is created. All one time Setup
// should be done now.
void MyEventSegment::Initialize()
{
    int ped = 110;
    int lld = 1;
    int uld = 100000;

    branchinit(Branch); // initializes the branch controller
    crateinit(Branch, Crate); //initializes crate

    camctl(Branch, Crate, Slot, 0, 9); //Clear Module
    camwrite16(Branch, Crate, Slot, 0, 23, 0xffff); // clear ctrl register bits
    camwrite16(Branch, Crate, Slot, 0, 19, 7); // set ctrl register bits
    camwrite16(Branch, Crate, Slot, 0, 17, 0); // select pedestal memory
    for(int i =0; i <16; i++) {
        camwrite16(Branch, Crate, Slot, i, 20, ped);
    }
    camwrite16(Branch, Crate, Slot, 1, 17, 0); // select low level disc. memory
    for(int i =0; i <16; i++) {
        camwrite16(Branch, Crate, Slot, i, 20, lld);
    }
    camwrite16(Branch, Crate, Slot, 2, 17, 0); // select upper level disc. memory
    for(int i =0; i <16; i++) {
        camwrite16(Branch, Crate, Slot, i, 20, uld);
    }
}

// Is called after reading data buffer
void MyEventSegment::Clear()
{
    // must use front panel clear
}

unsigned int MyEventSegment::MaxSize()
{
    return 16*4+2;
}

//Is called to readout data on module
DAQWordBufferPtr& MyEventSegment::Read(DAQWordBufferPtr& bufpt)
{
    if (camread16(Branch, Crate, Slot, 1, 6) ==0) return bufpt; // tests if data present
    bufpt = m_MyPacket.Begin(bufpt); //o pens new packet

    if(qtst(Branch)) {
        unsigned short tdcvalue = camread16(Branch, Crate, Slot, 0, 4);
        unsigned short channel = (tdcvalue >> 13) & 0xf;
        *bufpt = channel;
        ++bufpt;
        *bufpt = tdcvalue & 0xffff;
        ++bufpt;
    }
8.3 MyEventProcessor.h

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

class MyEventProcessor : public CEventProcessor
{
private:
  int Base; //base slot for data
  int nOurId; //ID tag to unpack

public:
  // Tests if a given packets Id = nOurId
  virtual Bool_t operator()(const Address_t pEvent, CEvent& rEvent,
      CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder);
  // constructor
  MyEventProcessor(int base, int Id);

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

8.4 MyEventProcessor.cpp

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

// Constructor: defines the Id to decode and Base slot to
// put data into
MyEventProcessor::MyEventProcessor (int base, int Id) {
  nOurId = Id;  // Id to unpack
  Base = base;  //Base slot # to store data
}
```
// Searches for the Id tag we want, asks to have it unpacked if
// the right tag is found.
Bool_t MyEventProcessor::operator()(const Address_t pEvent, CEvent& rEvent,
   CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder)
{
    TranslatorPointer<UShort_t> p(*(rDecoder.getBufferTranslator()), pEvent);
    CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);

    UShort_t nWords=*p++;
    nWords--;  // get number of words in the packet
    rAna.SetEventSize(nWords*sizeof(UShort_t));

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

void MyEventProcessor::UnpackPacket(TranslatorPointer<UShort_t> pEvent,
   CEvent& rEvent)
{
    UInt_t nPacketSize = *pEvent; // get number of words let to be read
    ++pEvent;++pEvent; // skip over Id info and Header words
    nPacketSize -=2; // adjust words remaining

    while(nPacketSize)
    {
        UInt_t nChannel = *pEvent; // get the channel number
        ++pEvent; // look at next word
        unsigned int value = *pEvent; // get ADC value
        rEvent[Base + nChannel] = value; // record ADC value
        ++pEvent; // increment pointer
        nPacketSize -=2; // subtract from packet size

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