9/25/2017

Support fror CAEN PHA DPP

For NSCLDAQ

Ronald Fox

Support fror CAEN PHA DPP

For NSCLDAQ

Introduction

CAEN DPP PHA firmware provides digital data acquisition for cases where the height of input pulses is proportional to the physical quantities being measured. This document describes the support provided for CAEN DPP PHA firmware under NSCLDAQ. It describes, as well, what is needed to integrate components of the data acquisition system that read CAEN DPP-PHA devices with other NSCLDAQ Readout components (e.g. VMUSBReadout).

The remainder of this document is organized as follows.

- The software is described in general terms.
- How the software is meant to be used alongside other CAEN software, specifically MCA2 and Compass ~~is~~ are described.
- A brief installation guide is provided that walks you through building the software from its distribution tarball to usable form.
- A sample set of configuration files is described and pointed to.
- Using the NSCL event builder to glue together events from multiple channels of the digitizer and other, non-CAEN devices is described.

How the software works.

NSCLDAQ provide several readout frameworks. Each of these frameworks was initially designed for use with a specific hardware interface. For example, VMUSBReadout is a framework designed to work with the WIENER/JTEC VMUSB controller.

One of these frameworks, SBSReadout, was initially designed for use with the SBS/Bit3 PCI/VME bus adapter. It actually is quite a bit more general than that. At the NSCL, for example it was tailored to be used with the XIA Pixie16 digital data acquisition system.

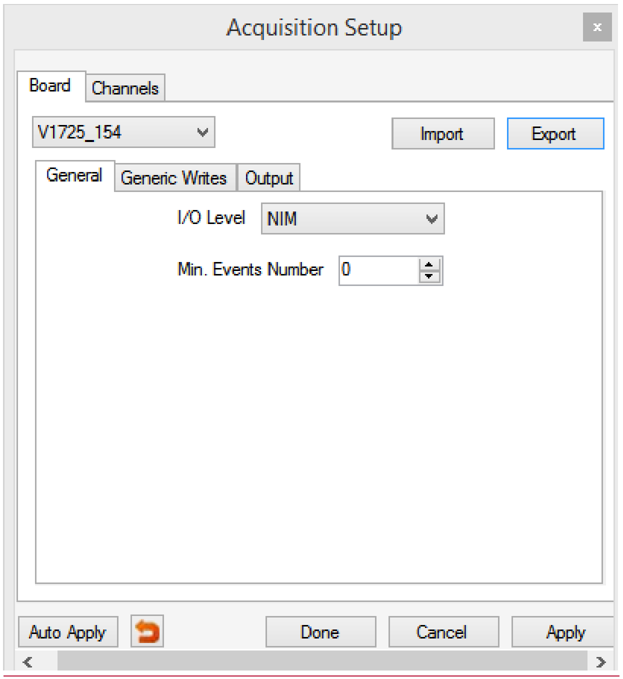
The SBSReadout framework is applicable whenever the hardware can specify some action trigger and user supplied code can respond to that trigger. SBSReadout is the basis for NSCLDAQ support of the CAEN DPP-PHA firmware. At present, support is provided for the CAEN xx725 and xx730 digitizers. Testing was done most extensively with a V1725 as this was the digitizer model that LSU purchased.

Using the software with MCA2

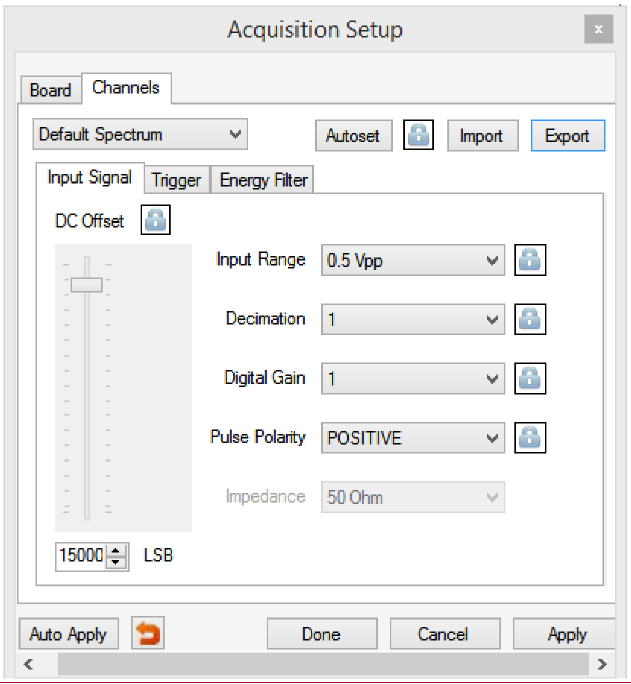
Formatted: Heading 1

Setting up a PHA DPP digitizer can be a bit challenging. CAEN provides a tool, MCA2 to assist with that process. The MCA2 manual also provides a step-by-step process for determining appropriate DPP parameters. Rather than re-invent that wheel, support for these digitizers under NSCLDAQ was written to be able to make use of the parameter description files MCA2 produces.

MCA2 produces two types of settings files. One type of file .cbs files provide board wide parameters. It is produced from the Acquisition setup window by selecting the Board tab as shown below and clicking the `Export` button:



The second type of settings file .ccs files describe channel specific settings and provide most of the DPP parameters. These are produced by seleting the Channels tab and clicking `Export` from the window below:



Default Spectrum

DC Offset: 15000 LSB

Input Range: 0.5 Vpp

Decimation: 1

Digital Gain: 1

Pulse Polarity: POSITIVE

Impedance: 50 Ohm

When setting up the CAEN DPP Readout program you'll need a board setup file for each digitizer board and a channel setup file for each channel. Fortunately in many cases, the same board setup file and channel setup file can be used for several boards and channels.

SBS Readout and Event Segments

SBS readout responds to a trigger by executing user code. The trigger condition is defined by user code as well. The code that responds to that trigger is organized into *Event Segments*. At its upper levels, the software provides the following classes:

- PHAEventSegment - provides a class that can readout a single module.
- PAHTrigger – provides a trigger class that can check the triggers on several modules.
- PHAMultiModuleEventSegment – provides a class that can perform "fair" readout on a list of modules. Fair readout, in this case means that if the rates are high no module will be starved for attention.

PHAEvent segment is built, in turn, on top of the CAENPha class which is a generic driver for CAEN Digitizers running the DPP-PHA firmware. This class is built on top of CAENPhaParameters and CAENPhaChannelParameters.

CAENPhaParameters, parses.cbs files (board level parameters) into a structure that is convenient for CAENPHa. Similarly, CAENPhaChannelParameters parses .ccs files (per channel parameters) into a vector of per channel parameter structs.

Formatted: Heading 2

Specifying a digitizer

Specifying a digitizer is a matter of:

- Creating a master configuration file for that digitizer card.
- Providing board level and channel level configuration files from MCA2 for the digitizer and its enabled channels.
- Instantiating a PHAEventSegment and, if you are using several digitizers adding that event segment to a PHAMultiModuleSegment.
- Adding the appropriate top level event segment to the Readout.

A worked example at the end of this part walks through this entire process.

The master configuration file is a file that is in Microsoft .ini format. https://en.wikipedia.org/wiki/INI_file describes the general form of this file.

Sections in this file are `[COMMON]` which provides module global parameters that are not provided by MCA2. Typically the keys that are important are:

Key	**Form**	**Comments**
OPEN	Type node [base]	Provides the connection parameters used to access the digitizer. This is a set of three white space separated words. The first word is either USB for a usb connection either to a desktop digitizer or a CAEN V1718 USB/VME bus bridge or PCI for CONET optical fiber cards. The second word is the CONET node number for optical fiber or 0 for USB. The last word is the base address if the connection is via a CAEN V1718 USB/VME bus bridge, 0 otherwise. For example OPEN USB 0 0x33330000 Open via V1718 for a module at 0x33330000 on the VME bus. OPEN PCI 1 0 Open CONNET node 0.
START_MODE	START_MODE how	Describes how the module will start taking data. It is important that this be set correctly to synchronize several digitizers that run on a common clock. Values are: **SOFTWARE** – software starts the module – only useful if running a single digitizer. **S-IN** – Module starts via a pulse on the S-IN input. A common S-In can be used to synchronize the start of several modules.

Formatted: Heading 2

Formatted: Font: Bold

Formatted Table

Formatted: Font: Bold

Formatted: Font: Bold

Formatted: Font: Not Bold

Formatted: Font: Not Bold

Formatted: Font: Not Bold, Underline

Formatted: Font: Bold

Formatted: Font: Bold

		TRIG1 – Module starts on the first trigger (which is not considered an event). Usually this is done with external triggers providing the start as for S-IN **GPI** – The GPI input (Desktop digitizers) is used to start data taking. Example for a single board test: START_MODE SOFTWARE
GLOBALXMLFILE	GLOBALXMLFILE path	File system path to the .CBS file MCA2 wrote for the board parameters. This file is an XML formatted file.

In addition to the `GLOBAL` section there can be zero or more channel sections. Channel section names are the channel number e.g. `0` for channel zero. The channel section has two keys:

Key	**Format**	**Comment**
enabled	Enabled TRUE | FALSE	The value `TRUE` means the channel is enabled. By default this is FALSE. Thus channels that don't have sections in the configuration file are defaulted to disabled.
ConfigXML	ConfigXML Path	Provides the path to the channel's configuration XML file. This is a .ccs file written by MCA2.

Here's a sample .ini file I have used:

```
[COMMON]

OPEN  USB 0 0x33330000
START_MODE SOFTWARE
GLOBALXMLFILE ../V1725_154.cbs

[0]
enabled TRUE
ConfigXml "../V1725_154 channel-0.ccs"
[1]

enabled TRUE
ConfigXML "../V1725_154 channel-0.ccs"
```

Formatted: Font: Bold
Formatted Table
Formatted: Font: Bold
Formatted: Font: Bold
Formatted: Font: Not Bold
Formatted: Font: Not Bold
Formatted: Font: Not Bold
Formatted: Font: (Default) Courier New, 11 pt
Formatted: Font: (Default) Courier New, 11 pt

I'm opening a digitizer via a V1718 USB/VME interface. The digitizer base address is 0x33330000. Only channels 0,1 are enabled. Since the signal shapes going into both channels are the same, I can use the same channel configuration file for both channels.

Adding a digitizer to readout and to the trigger

Formatted: Heading 2

The SBS readout provides a skeleton at $DAQROOT/skeletons/sbs. I suggest, however that you use the test subdirectory of the tarball as a skeleton instead as the Makefile there has been modified to pick up the needed libraries.

```
PHAMultiModuleSegment* mainSegment = new PHAMultiModuleSegment;   ①

PHAEventSegment*    seg1 = new PHAEventSegment("module1.ini", 1); ②
mainSegment->addModule(seg1);



PHATrigger* pTrigger = new PHATrigger;                            ③
pTrigger->addModule(seg1);

pExperiment->AddEventSegment(mainSegment);                        ④
pExperiment->EstablishTrigger(pTrigger);                          ⑤
```

Note that there are several definitions at the top of the Makefile that will need to be adjusted.

The Skeleton contains the Skeleton.cpp file. This must be edited to provide both the trigger and the top level event segment. Here's a code fragment:

① This creates a new multi module event segment. This will be used as the top level event segment.

② Creates a new event segment. The first parameter is the path to the top level .ini file for the module. The second parameter is a unique source id. The source id is used by the event builder to determine into which input fragment queue events from this digitizer will be inserted. The CAEN modules and firmware are designed so that data from one module are totally time ordered. Data read across multiple modules will, in general not come out time ordered. Placing different module data in different input queues allows the event builder to properly sort the data in to time order for the build stage.

③ This creates a new trigger object for DPP-PHA firmware modules. The trigger will fire when any module has data. This is a bit of a simplification but close enough to give an idea of how this works. The `addModule` tells the trigger to monitor for data in that module.

④ This line makes the top level event segment our multi module segment. When a trigger fires, the event segment will be asked to read data. The multimodule segment in turn will ask each individual module if it has data and will provide data from the first module. Which module is first is time varying so that no module is starved for attention at high rates.

⑤ Similarly this line of code establishes the DPP-PHA trigger as the experiment trigger. Thus when a run is active, the SBSReadout framework will poll our trigger and, when our trigger fires, read an event from the modules managed by the event segment.

The test subdirectory of the software distribution contains this complete example; code, Makefile, and top level .ini file. The top level of the software distribution contains the .cbs and .ccs files pointed to the .ini file.

Using the software with Compass

Formatted: Heading 1

As with MCA2, Compass can be used to interactively set up the digitizer and save the settings for production runs. Documentation of Compass is beyond the scope of this manual. Once you've setup your digitizer(s) properly for the input signal characteristics, save the project. This will produce an XML file called settings.xml in the .compass folder of your compass project.

```
#include <CompassTrigger.h>                                ①
#include <CompassMultiModuleEventSegment.h>                ②
#include <CompassEventSegment.h>                           ③
#include <CompassProject.h>                                ④
```

Formatted: Line spacing: single, Don't adjust space between Latin and Asian text, Don't adjust space between Asian text and numbers

Formatted: Font: (Default) Courier New, 11 pt, Font color: Auto

The NSCLDAQ support software can use this file as well to set up digitizers appropriately. A sample tailored SBReadout is available in the compasstest directory of your installation. Let's have a look at key parts of the modified skeleton file in that subdirectory. It assumes the settings.xml file has been copied from the compass project directory to this directory.

Several headers must be included to define classes we will need in our code:

① As with MCA2, the software knows that it must read when there is data available from any digitizer. The **CompassTrigger** class manages a set of boards to provide trigger status to the readout framework.

②, ③ Provide an event segment made up of several boards and one made of a single board respectively. Event segments are clases that understand how to prepare hardware for data taking and how to take data from that hardware.

④ The CompassProject class represents a compass configuration file which provides the configuration of one or more CAEN digitizer boards.

Next let's have a look at the code in the Skeleton's SetupReadout. This code creates event segments, bundles them into the multimodule event segment and uses the boards as contributors to the system trigger. Note that triggering is a complex (though this complexity is hidden from you). When data is available from a board, several events can be read. Only one event, however is provided to the framework. The trigger for that board remains true as long as data read has not yet been consumed.

① Each SetupReadout must begin with this call to the base class's method. This ensures that framework initialization is performed in addition to the code we've added.

② This creates a new CompassEventSegment to handle a board that is connected via USB to a VME interface. The board's VME base address is 0x33330000. The first parameter is a path to a compass configuration file that must have a description for a board that is connected physically in the manner described. The second parameter is the event

```
void
Skeleton::SetupReadout(CExperiment* pExperiment)
{
  CReadoutMain::SetupReadout(pExperiment);                        ①


  CompassEventSegment* pBoard1 =
    new CompassEventSegment("./settings.xml",
                           1,                                     ②
                           CAEN_DGTZ_USB,
                           0, 0, 0x33330000);

  CompassMultiModuleEventSegment *mainSegment =
    new CompassMultiModuleEventSegment;                           ③

  mainSegment->addModule(pBoard1);                                ④

  CompassTrigger* pTrigger = new CompassTrigger;                  ⑤
  pTrigger->addModule(pBoard1);


  pExperiment->AddEventSegment(mainSegment);                      ⑥
  pExperiment->EstablishTrigger(pTrigger);
```

builder source id the data from this board is tagged with. Data from an individual board is time ordered. Data from all boards, are not time ordered with respect to each other. By giving each board a unique source id and using the NSCL Event builder, data can be totally time ordered. Furthermore an appropriate coincidence window can group hits from several boards into coincident events.

```
③ In this single board example, we really don't need a multi-module
event segment. It's used here just to show how to use it to aggregate
data acquisition from several boards into a single object.
```

④ The idea is that for each board in the system, you create a board and then add it to the multi module event segments.

⑤ Similarly, in order to take data we need to create a trigger and add all of the boards to this trigger.

⑥ Finally the event segment and trigger are registered and become effective as soon as the run is started.

Installing the software

The software is distributed as a compressed tar file (tarball). The name of the tarball is DPP-PHA.tar.gz Unwrap the tarball in a directory under which you'd like the software installed:

```
tar czf DPP-PHA.tar.gz
```

This will create a directory: DPP-PHA in which the distribution lives. Edit the Makefile in that directory so that the definition of DAQROOT points to the top level directory of the NSCLDAQ installation. Build the software. You must be running a reasonably recent NSCLDAQ distribution. I recommend 11.2-006. The email describing where to get the tarball will also describe where you can find that.

```
make
```

To make the sample program, cd to the test subdirectory and edit the Makefile there to make the INSTDIR variable point at the top level directory of your NSCLDAQ installation.

```
make
```

To build the test program.

Using the NSCL Event builder.

Formatted: Heading 1

A prerequisite of the NSCL event builder is that all systems run on a common clock and must be reasonably synchronized. Reasonably synchronized means that the difference in clocks at t=0 (start of the run) is much smaller than the event building coincidence window. Note that if the difference is a constant, but known, the programs that feed the event builder can add or subtract a time offset from all timestamps. http://www.caen.it/servlet/checkCaenDocumentFile?Id=9137 is a CAEN technical report that describes how to achieve tight synchronization between CAEN digitizer boards. Synchronization between CAEN boards and non-CAEN system is, by necessity somewhat ad-hoc.

The event builder is documented at: http://docs.nscl.msu.edu/daq/newsite/nscldaq-11.2/c4465.htm It integrates well with the ReadoutGUI as calls from your ReadoutCallouts.tcl script.

The one thing that is a bit different about using DPP-PHA support is that usually a single readout produces data with a single source id. The DPP-PHA support produces a source id for each of the DPP-PHA modules in the setup. Fortunately the event builder and its feeders system is able to deal with that as it was originally anticipated that XIA/DDAS would need this.

NSCL event builder is usually started by using the NSCL ReadoutGUI. The ReadoutGUI provides the ability to extend its functionality via the use of a ReadoutCallouts.tcl Tcl script. This capability is described in general terms at http://docs.nscl.msu.edu/daq/newsite/nscldaq-11.2/x4050.html which is part of the general documentation of the NSCL Readout GUI for NSCLDAQ-11.x (Beginning at http://docs.nscl.msu.edu/daq/newsite/nscldaq-11.2/c3603.htmlhttp://docs.nscl.msu.edu/daq/newsite/nscldaq-11.2/c3603.html).

Integrating the Event builder with the ReadoutGUI is described in the event builder manual at http://docs.nscl.msu.edu/daq/newsite/nscldaq-11.2/x4476.html

Let's look at how to modify the example in that web page:

```
package require evbcallouts

::EVBC::useEventBuilder

proc OnStart {} {
  ::EVBC::initialize -restart false -glombuild true -glomdt 123
}

EVBC::registerRingSource tcp://localhost/sidet0 {} 0 "Si Dets" 1 1 20 0
```

First let's look at the EVBC::Initialize call in OnStart. This can be tailored to your need by modifying the value of the -glomdt parameter to be the coincidence window desired for your experiment. You may also want to specify the -destring option whose value is the name (no URL) of the ring buffer into which the event builder will insert its data. If you want the event builder to restart each run, -restart can be true

There must be one call to EVBC::registerRingSource for each readout program you have sending data into the event builder. This call arranges for a program called ringFragmentSource to take data from the ringbuffer of a readout program, wrap it event builder headers and ship those data to an event builder running in the localhost (that was arranged by EVBC::initialize).

The parameters are in order:

1. The URL Of the ring buffer from which the ringFragmentSource program will take data.
2. An optional path to a shared library that knows how to extract time stamp from events. 11.2 event format provides for an event *body header* which contains the timestamp and source id of an event in a predictable place. Therefore this parameter should be left empty.
3. A Tcl Formatted list of the source ids that will be encountered in the data from that ring. It is an error for the ringbuffer to have an event from a source id that is not in this list. This should be a list of the source ids in all PHAEventSegment's you instantiated in the Readout Program that's submitting data to the ringbuffer specified by the first parameter.
4. A string that describes the data from this source. This is just displayed on the monitor GUI for the event builder in the part of that display that describes the event source statuses.
5. A flag that is nonzero (true) if the data is guaranteed to have body headers. 11.2 event data from SBSReadout with PHAEventSegemtns always has a body header. Therefore this should be true.
6. True if the ringFragmentSourcde should be restarted for every run. If true, the ring fragment source exits after encountering and submitting an end run item. This should normally be true. The event builder Readout Callouts library will restart event sources that require restart at the beginning of each run.
7. Event builders can be hierarchically composed. How to do that is beyond the scope of this document. If that is done, an event source may have more than one end run item at the end of a run, one for each source. This value is the maximum number of seconds to wait for all end run items to be observed at the end of a run. It is only important in oneshot mode. The ringFragmentSource counts begin run items. It expects the same number of end run items. If the timeout expires after seeing at least one end run item but before seeing all end run item it assumes that one of the event sources died during the run and does not wait any longer before exiting.
8. This is a signed integer value that is added to the timestamps in each item when constructing the event builder header. Note that the original event is not modified. Each event is wrapped with a header that, among other things, includes the timestamp the event builder should use for ordering and combining fragments. This timestamp is constructed by *rawstamp + offset* where *rawstamp* is the timestamp reported by the event and *offset* is this parameter. This can be used to compensate for time offsets at start up that are fixed but large enough that the event build window is of questionable validity (e.g. offsets on the order of or larger than the event build window).