Specifications for the bias voltage control software API for Hall D tagger microscope

Hovanes Egiyan, Richard Jones

April 29, 2013

Abstract

We specify the interface required to integrate the voltage control of the GlueX tagger microscope detector into the experimental control system of Hall D.

1 Introduction

The tagger microscope detector consists of five columns of 100 rows of scintillators that will be read out using silicon photodetectors (SiPMs). The bias voltage on each of the SiPMs needs to be individually adjusted with a precision of 0.01 Volts. The control points for the voltage supply to the SiPMs are divided into groups of 30 channels with a total of 17 groups. Each group is controlled using a custom designed board that can set and report values of the different parameters for each SiPM channel. The remote communication with the board can be done through an RG45 Ethernet port using custom-designed Transport Layer protocol. Each control board is identified by an address set using a jumper, and each channel is identified by a geographical address header word in the response packets sent by the device to the host.

The programming of the firmware on the control boards has been done by the University of Connecticut group, and in order to be able to integrate the bias voltage control and monitoring into the EPICS-based controls framework of Hall D an interface layer needs to be developed to be used by the EPICS Input/Output Controllers (IOCs) running on a regular Linux-based computer. Within the EPICS framework different parameters of each voltage channel will have their corresponding EPICS variables. The set-point variables that write to the control boards will be processed (and thus communicate with the board) whenever the desired value is changed by the user, while the variables that obtain their values from the boards will need to be updated at about 1 Hz frequency. Note that the initial version of the API will not contain some of the parameters representing the actual values, such as the voltage and current readback. The design of the control board will be modified at a later time to include the readback values. This document specifies the requirements for such a library assuming that the time for sending a single requests to a control board and receiving a response from it over a local network is on average under one millisecond during the operations with the full configuration. It does not assume though that the values of parameters obtained from the control boards necessarily match the actual values of these parameter at a time within one millisecond of the readout time. This assumption merely allows the EPICS software layer not to consider possibilities of backing up the queues for communicating with the control boards.

There will also be a set of parameters related to the microscope control boards themselves. Each board will have a read-only temperature value as well as a status word to indicate the health of the control board and its firmware.

2 List of parameters

Table 1 lists the parameters that will be controlled and monitored by EPICS for every voltage channel. A brief description for each parameter is given in the second column. The column "Access by EPICS" indicates whether the parameter is expected to be read, written, or both by EPICS. The **status** parameter should indicate the state the channel is in, such as *On*, *Off*, *Ramp-up*, *Ramp-down*, *Tripped*, *Over-temperature*, etc. The **enable** parameter can be written by EPICS to turn on and off the channel, and it also can be read by EPICS to determine what the most recent EPICS request for enabling or disabling that channel was. Note that it provides us with different information from what is reported in **status** parameter which reports the actual state of the voltage channel. The **status** word will be read from the board for each channel and represent the status of the channel and will indicate if there is any problem with the channel. The API will provide a translation for each of these states.

Table 2 shows the parameters pertaining to the microscope control boards.

Short Name	Brief Explanation	Units	Access by EPICS
v_sp	Voltage setpoint	V	Read/Write
v_max	Maximum allowed voltage	V	Read/Write
temp_rb	Temperature reading	^{0}C	Read
ramp_up	Ramp-up rate	V/s	Read/Write
ramp_dn	Ramp-down rate	V/s	Read/Write
status	Channel status	N/A	Read
enable	Enable/Disable channel	N/A	Read/Write
gain_mode	Preamp gain mode	N/A	Read/Write

Table 1: List of the parameters than need to be controlled/monitored by EPICS for each microscope voltage channel.

Short Name	Brief Explanation	Units	Access by EPICS
temp_rb	Temperature reading	^{0}C	Read
status	Board status	N/A	Read

Table 2: List of the parameters than need to be monitored by EPICS for each microscope control board.

3 List of functions

In order to access and modify the parameters in Table 1 we need to develop a library of functions that will be used by a program running on a remote host and communicating with the control boards over a local network. This program will serve the EPICS variables corresponding to different parameters using ChannelAccess protocol. The most commonly used programming languages for developing an EPICS ChannelAccess servers are C and C++, therefore the set of functions needs to be easily usable by C and C++ codes.

The methods that are required in the API can be divided into three types. The first type of the functions will be accessing and modifying the parameters listed in Table 1 that are related to a particular bias voltage channel. These functions are included in the C++ header file presented in Sec. 5.1. The second type of functions would change or report the status of the communication between the server and individual boards (or ports). The list of functions operating on an individual control board can be found in the sample C++ header file in Sec. 5.2. The third type can represent the functions that refer to the API as whole and are not related to any channel or board. All details of the communication protocol will be hidden from EPICS support allowing EPICS to open communication ports for each control board, to determine the list of channels available on each port, and to send requests for individual channel and receive responses using function calls. EPICS will call these functions asynchronously, that is the requests for each board will be queued, and when it is time for the request to be processed a callback function is processed in a separate thread. In general, the calls to the communication ports for the control boards will be made from multiple threads at the EPICS support level, therefore the API implementation should have its own mutual exclusion scheme such that there are no collisions between the calls from multiple threads and that the dead-time of the communication ports due to locking of the shared resources on the server side is not prohibitively high. EPICS support level will not attempt to prevent accesses to the communication ports from multiple threads and will threat all methods and functions in the API as re-entrant and thread-safe.

4 Summary

In this document we specified the requirements for the API that needs to be implemented to integrate the bias voltage control into the Hall D experimental control system. This paper defines a list of parameters that should be represented in the firmware as well as a set of functions that are needed for EPICS device support (see the Sec. 5). The software library will need to be compiled and run on a Linux-based system, and will only require external libraries that are part of the major Linux distributions.

5 Appendix

5.1 Header file example for a class corresponding to the bias voltage channels

```
/*
 * UConnBiasChannel.hh
 *
 *  Created on: Apr 28, 2013
 *      Author:
 *
 *      This class provides interface that EPICS support will use to monitor
 *      and control bias voltage channels on UCon Tagger Microscope detector.
 *      All methods, including accessotrs, need to be thread-safe since they
 *      may be executing simultaneousely on multiple threads.
 *
 */

#ifndef UCONNBIASCHANNEL_HH_
#define UCONNBIASCHANNEL_HH_

#include <stdint.h>

#include <iostream>
#include <string>
#include <vector>
#include <map>

class UConnBiasChannel {
protected:
// Prevent copying boards
UConnBiasChannel( const UConnBiasChannel& chan );
UConnBiasChannel& operator=( const UConnBiasChannel& chan );

public:

// Constructor of a channel with geo-address chanAddress on boards with an
// address boardAddress
UConnBiasChannel( const std::string boardAddres, const std::string chanAddress );
// Destrcutor
~UConnBiasChannel();

// Set and get the setpoints for bias voltage
void SetVoltageSetPoint( double voltage );
double GetVoltageSetPoint();

// Set and read the firmware limit on bias voltage
void SetMaxVoltage( double maxVoltage) ;
double GetMaxVoltage();

// Set and read the ramp-up rate for bias voltage
void SetRampUpRate( double rate );
double GetRampUpRate();

// Set and read the ramp-down rate for bias voltage
void SetRampDownRate( double rate );
double GetRampDownRate();

// Read the 32-bit status word for this channel
uint32_t GetStatus();

// Enable, disable this bias channel
void Enable();
void Disable();
bool IsEnabled();

// Read the temperature value
double GetTemperature();

// Get the board address to which this channel belongs to
std::string GetBoardAddress();
// Get the geographical address of this channel
std::string GetChannelAddress();
};

#endif /* UCONNBIASCHANNEL_HH_ */
```

5.2 Header file example for a class corresponding to the control boards

```
/*
 * UConnCtrlBoard.hh
 *
 *  Created on: Apr 28, 2013
 *      Author:
 *
 *      This class provides interface that EPICS support will use to monitor
 *      and control the control boards on UCon Tagger Microscope detector.
 *      All methods, including accesssors, need to be thread-safe since they
 *      may be executing simultaneousely on multiple threads.
 */

#ifndef UCONNCTRLBOARD_HH_
#define UCONNCTRLBOARD_HH_

#include <stdint.h>

#include <iostream>
#include <string>
#include <vector>
#include <map>

#include "UConnBiasChannel.hh"

class UConnCtrlBoard {

protected:
// Prevent copying boards
UConnCtrlBoard(const UConnCtrlBoard& board);
UConnCtrlBoard& operator=(const UConnCtrlBoard& board);

// Map that keeps track of the channels served by the board
// Key is the address, the address is the address of the
// corresponding UConnBiasChannel object instance.
std::map<std::string,UConnBiasChannel*> uccbChannelMap;

// This static member is a map that keep record of all control board
// object instances. It is accessible through a class method to ensure
// the thread safety.
static std::map<std::string,UConnCtrlBoard*> uccbBoardMap;

public:

// Constructor. Open communication for board specified by "address"
// Create all channel objects on that board and fill the channel map. It also
// registers the create board in the map of boards. Throws an
// exception if fails to create the object.
UConnCtrlBoard(std::string address);

// Destructor. Closes this communication channel. Removes the address of
// this board from the map of the boards.
~UConnCtrlBoard();

// Reset the communication channel
int Reset();

// Method to check if communication is active
bool IsConnected() ;


// Return the measured temperature of the board
double GetTemperature();

// Return the 32-bit status word for this board
uint32_t GetStatus();

// Return the number of channels this board is serving
unsigned GetNumberOfChannels();

// Return the address of the board
std::string GetAddress();

// Create and return a vector with pointer to all channels
// served by this control board.
std::map<std::string,UConnBiasChannel*>GetChannelMap();

// Translate the status word to a list of human readable messages
```

```
static std::vector<std::string>  StatusString(unsigned long status);

// Get the map of pointers to all active boards.
static std::map<std::string,UConnCtrlBoard*>GetBoardMap();
};

#endif /* UCONNCTRLBOARD_HH_ */
```